JOB RECOMMENDATION SYSTEM - DA331 FINAL REPORT

Borra Lakshya Nabeel Khan

***Abstract*—In the landscape of today's overwhelming data, recommendation systems have evolved as essential tools, particularly in the realm of job recruitment. They leverage machine learning to streamline information, aiding users in navigating vast job databases. However, in the current scenario, job-seeking platforms often fail to personalize recommendations based on individual skills and profiles, especially for college graduates transitioning into fields like IT.**

This gap has prompted a critical examination of recommender systems that assesses users' resumes, analyzing qualifications, soft and hard skills, and past projects to provide tailored job suggestions and improve resume scores. These systems aim not only to recommend jobs but also to guide users in skill enhancement for better job matches.

***Index Terms*—Job Recommendor systems, Recommendation systems, Candidate Hiring, HR Research**

I. Introduction and Motivation

The evolution of job recommendation systems has been catalyzed by the immense growth of online job-seeking platforms and the need for more efficient and personalized solutions. The advent of AI and machine learning has spurred innovation, propelling these systems beyond traditional methods.

One notable challenge lies in the vast disparity between job descriptions and candidates' resumes. While these systems have made strides in matching skills and qualifications, there's a pressing need to refine algorithms to interpret implicit behaviors, fill in sparse user profiles, and capture the nuances of evolving career trajectories.

Furthermore, the ethical considerations surrounding job recommendation systems are gaining prominence. Fairness and bias mitigation within these algorithms are critical, particularly in ensuring equitable opportunities for all candidates, regardless of demographic factors or implicit biases.

The global impact of the COVID-19 pandemic has reshaped the employment landscape, underscoring the importance of adaptive job recommendation systems. Remote work opportunities, changing industry dynamics, and the rise of the gig economy have prompted a reevaluation of traditional job-seeking norms.

Various studies have explored enhancing job recommendation systems by considering skills as crucial factors for matching job descriptions to user profiles. Some research integrates skills as keywords, while others focus on user-job dyadic data for recommendation modeling. Additionally, there are approaches utilizing job history but overlooking associated skills, and studies addressing feature selection optimization and job clustering for e-recruitment systems.

Additionally, the integration of natural language processing (NLP) techniques into these systems offers promise in enhancing the contextual understanding of both job descriptions and candidate profiles. Leveraging NLP for sentiment analysis, context extraction, and semantic matching presents an avenue for more refined and accurate job recommendations.

As the job market continues to evolve, the amalgamation of cutting-edge technology, ethical considerations, and a deeper understanding of evolving career trajectories will define the next phase of job recommendation systems. These systems not only aim to streamline job searches but also play a pivotal role in shaping the future of work by facilitating better job-person fit, promoting career growth, and fostering a more inclusive job market.

Job recommendation systems need to adapt to changing user preferences and evolving job criteria. The dynamic nature of preferences requires these systems to be context-aware. Both jobs and candidates possess various attributes crucial for matching. Skills serve as identifiers of talent, guiding job preferences.

Candidate profiles evolve over time, hinting at their motivations and preferences as they progress in their careers. Matching candidates to jobs involves a complex amalgamation of attributes, making rule-based systems inadequate due to their inability to capture nuanced patterns in the data.

The increasing reliance on the internet for job hunting has led to overwhelming information on job-seeking websites. The process is inefficient as these platforms merely display information without personalized recommendations. An improved recommendation algorithm based on item-based collaborative filtering is proposed to address this issue.

The aim is to provide personalized services to students, aiding them in swiftly finding suitable jobs. Despite user profile limitations, future studies aim to address this by incorporating implicit behaviors into the recommendation matrix.

In conclusion, job recommendation systems are indispensable in today's expansive job market, particularly in rapidly growing sectors like IT services. They offer a data-driven, personalized approach to both recruiters and job seekers, ensuring better job matches, streamlined career transitions, and growth opportunities.

II. PROBLEM STATEMENT

In today's dynamic job market characterized by an exponential surge in available job postings across various online platforms, a fundamental challenge persists—the overwhelming influx of information inundating job seekers. The colossal volume of job listings on these platforms poses a significant hurdle, leaving users to navigate through an extensive array of opportunities often lacking personalization or tailored recommendations. Moreover, for college graduates or individuals transitioning between domains, the task of discerning ideal job matches becomes further convoluted, leading to time-consuming and inefficient job searches.

A critical gap emerges in the existing job recommendation landscape, primarily driven by platforms that predominantly display recruitment information to users, failing to incorporate individual user profiles and skill sets adequately. This limitation necessitates a profound reevaluation of the efficacy of current recommendation systems, particularly in addressing the personalized needs of users.

Present systems often overlook the nuanced attributes of a candidate's resume, neglecting to consider qualifications, soft and hard skills, and past projects that form the core of a user's profile. As a consequence, the recommendations lack the essential tailoring required to match individuals with suitable job opportunities.

Furthermore, amidst the rapid evolution of technology and the burgeoning interest in fields like Information Technology (IT), many individuals seek guidance on aligning their burgeoning skill sets with specific job roles. The existing job recommendation systems often fall short in addressing this critical need, as they predominantly focus on users' domain interests without robustly considering the complete spectrum of their skills and unique profiles. This gap highlights the dire need for an innovative job recommendation system capable of not only curating job options but also analyzing and leveraging individual skill sets, past experiences, and career trajectories to provide tailored and impactful recommendations.

The exploration of recommendation systems focuses on three core methods—content-based recommendation systems, memory-based collaborative filtering, and model-based collaborative filtering—highlighting the project's reliance on the latter. Each approach entails distinctive mechanisms, yet the choice of employing a model-based collaborative filtering framework for this project is specifically emphasized.

Content-based recommendation systems are founded on user profiles and item analysis, leveraging attributes like user interests and skills to drive recommendations. This model relies on precise user profiles for accuracy. Utilizing libraries such as 'sklearn.feature-extraction.text.CountVectorizer()' in Python facilitates the creation of intermediate representations (IMRs) based on textual data, employing vectors to represent user and item profiles. However, the reliance on hand-engineered features and limited expansion of user interests pose drawbacks, impacting the system's recommendation precision.

Memory-based collaborative filtering, split into user-based and item-based approaches, operates on the premise of users with similar tastes favoring analogous items. This method utilizes similarity functions such as cosine similarity, Euclidean distance, Jaccard similarity, and Pearson correlation to measure closeness between users or items. While simpler for implementation, this method demands significant computational power, affecting system performance.

In contrast, model-based collaborative filtering circumvents the need to load entire datasets into system memory. Employing machine learning techniques like clustering, neural networks, and matrix factorization, this approach avoids performance issues associated with large datasets. Matrix factorization, notably popularized after the Netflix Prize Competition, characterizes user and item attributes through vectors, predicting user ratings for items based on their preferences.

The corpus of work that currently exists primarily concentrates on the company's viewpoint and attempts to pair applicants with particular positions. Recommendation algorithms, including collaborative systems and the Minkowski distance formula, have been the subject of several experiments. But these algorithms typically work best in scenarios where there is a wealth of available data, which helps employers choose candidates more effectively than it helps job seekers find the best position for them. Content-based recommendation systems have surfaced as a potential solution, particularly for handling "cold start" problems in situations with little data. Scholars have emphasized that content-based systems are superior, particularly when compared to other systems for job recommendations.

In recommender systems, picking an algorithm is not the only step in the process of making an accurate forecast. The importance of text vectorization and similarity functions in this field has been underlined by authors. One prominent text feature selection method is the tf-idf approach, and the preferred similarity measure is cosine similarity, which shows efficacy not only in system recommendations but also in comparing phrase or paragraph similarities. Cosine similarity is superior to other similarity metrics, such as Euclidean, Jaccard, and cosine, according to studies comparing them. This is especially true when using multiple stemming types.

There are still problems in the field of employment recommendation systems, despite these developments. Numerous systems are designed with corporate interests in mind, which may result in job recommendations that aren't applicable for certain people. The usefulness of collaborative recommendation systems for individual job searchers is limited due to their heavy reliance on aggregated user data. Furthermore, several systems necessitate subscription charges or logging in, which results in spam email redirection. This emphasizes the need for more user-friendly and safe platforms. Although content-based recommendation systems have demonstrated potential, there are still implementation and security holes, which calls for more thorough methods to be used in system design and development.

Essentially, even with recent advances in similarity metrics and recommendation algorithms, a significant percentage of current systems continue to be biased toward business goals and ignore the user's job search history. Although the emphasis on content-based systems is encouraging, more work needs to be done on security, user-centered design, and implementation to create job recommendation platforms that are more efficient and convenient for users.

III. ARCHITECTURAL DETAILS

A. TEXT PREPROCESSING AND ENHANCEMENT TECHNIQUES:

The recommendation system integrates text preprocessing methods like Porter Stemmer and stop words removal. Porter Stemmer reduces words to their root forms, enabling easier comparison, while eliminating stop words streamlines the system by discarding non-informative terms.

PORTER STEMMER:

The Porter Stemmer is a widely used algorithm in natural language processing (NLP) that operates on English words to reduce them to their base or root form, known as the stem. The process of reducing words to their root form is called stemming.

Developed by Martin Porter in the 1980s, the Porter Stemmer follows a set of rules or algorithms that systematically trim suffixes from words, aiming to convert them into their base forms. The algorithm is rule-based and applies a sequence of linguistic rules to strip away common word endings.

Here's an overview of how the Porter Stemmer algorithm works:

Tokenization:
The text is initially broken down into individual words, known as tokens. This tokenization step prepares the words for the stemming process.

Application of Rules:
The Porter Stemmer applies a series of rules that define how to remove suffixes from words. The rules are designed to handle various word endings and normalize words to their root forms. For example: Rule application to handle plural forms: Words like ”cats” become ”cat”. Rule application for verb endings: Words like ”running” become ”run”.

Algorithm Iteration:
The algorithm applies a series of steps in a sequence to modify words. These steps are not applied in a fixed order; rather, they follow a set of prioritized rules to transform words.

Iterative Trimming:
The algorithm proceeds iteratively, trimming suffixes until it finds a match for one of its predefined rules. The goal is to reduce the word to its stem by removing suffixes that denote tense, plurality, or other variations.

Termination Condition:
The stemming process continues until the word can no longer be modified by any of the defined rules. At this point, the algorithm stops, and the resulting form is considered the stem of the word.

It's important to note that stemming using the Porter Stemmer does not always result in dictionary words. Stemming primarily aims to standardize words to their root forms, often sacrificing linguistic accuracy for simplicity and efficiency.

While the Porter Stemmer is a popular and widely used stemming algorithm, it's not without limitations. Stemming may sometimes produce stems that are not entirely accurate or meaningful in all contexts. Despite its limitations, the Porter Stemmer remains a foundational tool in natural language processing tasks like information retrieval, search engines, and text mining.

STOP WORDS :

In natural language processing (NLP), stop words refer to commonly used words that are often filtered out or excluded from text processing and analysis. These words typically occur frequently in language but often carry little or no specific semantic value. Examples of stop words include ”the,” ”and,” ”is,” ”in,” ”of,” etc. The goal of removing stop words is to reduce the dimensionality of text data and focus on more meaningful words that carry significant information.

Implementation of Stop Words Removal:

1. Identifying Stop Words:
Stop words are usually defined in predefined lists specific to a language. Libraries in various programming languages (such as NLTK for Python, SpaCy, or Gensim) often provide pre-built lists of stop words. These lists can be customized based on the specific application or domain.

2. Tokenization:

Before removing stop words, the text is tokenized, splitting it into individual words or tokens. This step ensures that each word can be examined independently.

3. Filtering Stop Words:

The identified stop words are then removed from the tokenized text. This step involves iterating through the tokens and eliminating any words that match the stop word list.

4. Preprocessing:

Stop words removal is typically part of a series of preprocessing steps in NLP. Other preprocessing steps might include lowercasing all words, removing punctuation, lemmatization (reducing words to their base form), or stemming.

B. MULTI-LAYERED RECOMMENDATION SYSTEM ARCHITECTURE:

The recommendation system comprises a multifaceted architecture, divided into three layers: the Applicant-Centric Layer, focused on aligning job listings with applicant requirements; Non-Technical Skill Evaluation, filtering based on interpersonal abilities and soft skills; and Technical Skill Assessment, evaluating technical proficiencies for elite candidate selection.

C. HOLISTIC APPROACH TO RECRUITMENT:

The system adopts a holistic approach, bridging the gap between job seekers and hiring teams. It offers constructive feedback to rejected applicants, empowering their growth and offers a comprehensive evaluation, including non-technical skills critical for success in job roles.

D. CONTINUOUS DATA ENRICHMENT:

Continuous data enrichment is crucial for the model's accuracy and relevance. Regular updates and enriching the dataset with new job offerings ensure optimal system performance, maintaining precision in recommendations.

E. TECHNICAL METRICS:

Cosine Similarity, a similarity metric, gauges item comparability based on angle determination. TF-IDF, another metric, assesses a term's importance in a document by considering its frequency in that document compared to its frequency across the entire corpus.

COSINE SIMILARITY:

Cosine similarity is a metric used to measure the similarity between two non-zero vectors. It determines the cosine of the angle between these vectors, indicating how closely they align in a multi-dimensional space. This method is commonly employed in various fields, including information retrieval, text analysis, recommendation systems, and machine learning, to compare the similarity between documents, texts, or any two entities represented as vectors.

Mathematical Concept:

Given two vectors A and B, the cosine similarity S between them is calculated using the dot product of the vectors and their magnitudes:

Here's a step-by-step breakdown of the calculation:

Dot Product :

The dot product is computed by taking the sum of the products of corresponding elements of the vectors.

Magnitude :

The magnitude (or Euclidean norm) of a vector A is the square root of the sum of squared elements.

Cosine Similarity (S):

The cosine similarity is obtained by dividing the dot product by the product of the magnitudes of the vectors.

Interpretation:

The cosine similarity value ranges between -1 and 1. A value closer to 1 indicates a higher similarity or greater alignment between the vectors. A value closer to -1 suggests dissimilarity, and 0 implies orthogonality (perpendicularity) between the vectors.

TF-IDF (Term Frequency-Inverse Document Frequency):

TF-IDF (Term Frequency-Inverse Document Frequency) is a numerical statistic widely used in information retrieval and natural language processing (NLP) to evaluate the importance of a term within a document relative to a collection of documents. It reflects how relevant a term is to a specific document in a corpus.

TF (Term Frequency):

Term Frequency measures the frequency of a term within a document. It signifies how often a term occurs in a document, typically normalized by the total number of words in the document to prevent bias towards longer texts:

TF(t,d) = (Total number of terms in document d) / (Number of times term t appears in document d)

IDF (Inverse Document Frequency):

Inverse Document Frequency evaluates the significance of a term across a collection of documents (corpus). It helps in highlighting terms that are rare in the entire corpus but crucial in specific documents:

IDF(t) = log(Number of documents containing term t) / (Total number of documents in the corpus)

TF-IDF Calculation:
The TF-IDF score for a term t in a document d is computed as the product of TF and IDF:

TF-IDF(t,d)=TF(t,d)×IDF(t)

Interpretation:
High TF-IDF scores signify that a term is both frequent within a specific document and relatively rare across the entire corpus. These terms are considered more important or distinctive for that particular document. Low TF-IDF scores indicate that the term is either common across all documents or infrequent within the document itself.

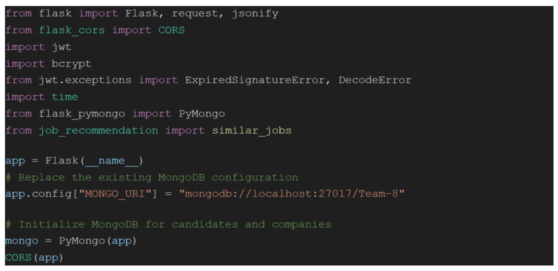
IV. METHODOLOGY AND EXPERIMENTS / DEMO

A. *DATA SOURCES AND SITES*

In the pursuit of constructing a comprehensive Job Dataset for research purposes, we combined multiple resources from Kaggle and conducted supplementary Google searches. Additionally, we collected an array of resume data from our network of acquaintances. It is noteworthy that certain datasets encompass supplementary information, notably location and salary columns, which serve the purpose of facilitating tailored job recommendations aligned with applicant preferences. In cases where these particular columns remain unpopulated, our model is designed to employ a hierarchical approach, thus prioritizing job recommendations based on inherent job attributes rather than specific location and salary requirements.

B. *TECH STACK FRONT-END EXPLANATION*

Imports and Flask server setup:

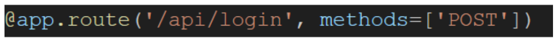


```
from flask import Flask, request, jsonify
from flask_cors import CORS
import jwt
import bcrypt
from jwt.exceptions import ExpiredSignatureError, DecodeError
import time
from flask_pymongo import PyMongo
from job_recommendation import similar_jobs

app = Flask(__name__)
# Replace the existing MongoDB configuration
app.config["MONGO_URI"] = "mongodb://localhost:27017/Team-8"

# Initialize MongoDB for candidates and companies
mongo = PyMongo(app)
CORS(app)
```

Importing necessary libraries for building a Flask web server and implementing the authentication and server routes for the job recommendation site. CORS enables Cross-Origin Resource Sharing to handle requests from different domains, since frontend runs on port 3000 and flask server runs on port 5000. jwt and bcrypt are packages used for user authentication with JSON Web Tokens and password hashing to store encrypted password to MongoDB. time is employed for expiration timestamp when encoding jwt tokens. Flask-PyMongo facilitates MongoDB integration with the Flask application, an abstraction for connecting with the mongodb instance and provides apis for flask developers. similar-jobs is imported from job-recommendation to provide job recommendation functionality, explanation to which is covered later. Initial step is to initializing the Flask app (app) and configuring it to use MongoDB with the specified URI. Also create a PyMongo instance (mongo) to interact with the MongoDB database, and also enable cors for frontend cors permissions.

Authentication Endpoints:

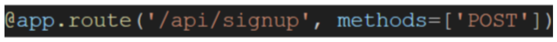
Login Endpoint:



```
@app.route('/api/login', methods=['POST'])
```

This endpoint handles user login requests. Retrieves the username and password from the request JSON. Checks whether the provided credentials match existing candidate or company records in the MongoDB. If authentication is successful, a JWT token is generated and returned else status code 401.

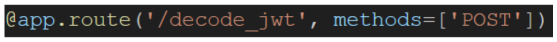
Signup Endpoint:



```
@app.route('/api/signup', methods=['POST'])
```

Manages user signup requests. Validates the provided data, checks for existing usernames, and determines user type. Hashes the password using bcrypt. Inserts the new user data into the appropriate MongoDB collection, 2 collections, candidates and companies and returns a JWT token, else 400 incase of invalid user type or 409 incase of existing user.

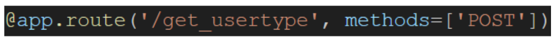
Decode JWT Endpoint:



```
@app.route('/decode_jwt', methods=['POST'])
```

Decodes and verifies a JWT token provided in the request. Returns the decoded username if the token is valid. Handles potential exceptions, such as an expired or invalid token, in that case it returns 401.

Get User Type Endpoint:

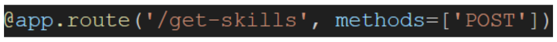


```
@app.route('/get_usertype', methods=['POST'])
```

Determines the user type (candidate or company) based on the provided username. Returns the user type or an error message, else 404 when user not found.

User Profile Management Endpoints:

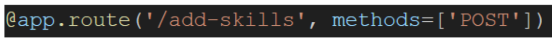
Get Skills Endpoint:



```
@app.route('/get-skills', methods=['POST'])
```

Retrieves skills associated with a given username from the MongoDB database. Returns the skills or an error message, 404 in case user not found.

Add Skills Endpoint:

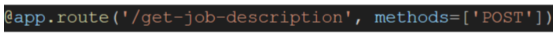


```
@app.route('/add-skills', methods=['POST'])
```

Adds or modifies skills associated with a given username in the MongoDB database. Returns a success message or an error message, else 404 when user not found.

Company Profile Management Endpoints:

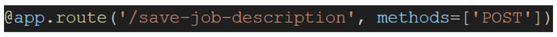
Get Job Description Endpoint:



```
@app.route('/get-job-description', methods=['POST'])
```

Retrieves the job description associated with a company username from the MongoDB database. Returns the job description or an error message, 404 in case company not found.

Save Job Description Endpoint:



```
@app.route('/save-job-description', methods=['POST'])
```

Saves or updates the job description associated with a company username in the MongoDB database. Returns a success message or an error message, 404 in case company not found.

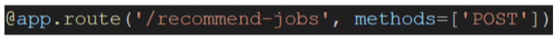
Job Recommendation Endpoint:

Calls the similar-jobs function from job-recommendation.py to recommend jobs based on candidate skills. Returns recommended jobs as a json object, else a 400 status code in case, skills not provided.

Additional Details to Server code:

Web Framework: Flask is chosen for building the web server due to the Proof of concept using python, specifically a .ipynb script. MongoDB Integration: Flask-PyMongo simplifies the integration of Flask with MongoDB, enabling seamless database operations. We push new companies and candidates in their corresponding collections, also do note, that we have taken a pre existing database from kaggle and other resources. JWT Authentication: Flask provides a

straightforward way to implement user authentication using



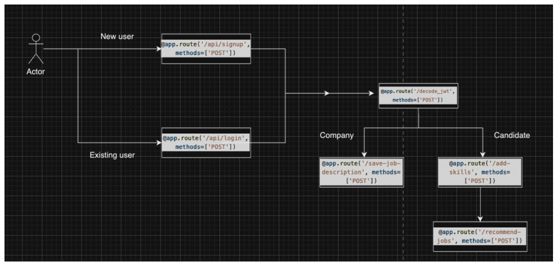
```
@app.route('/recommend-jobs', methods=['POST'])
```

JWT tokens, used to maintain and segregate user sessions. This prevents a user simply modifying the url to switch as a different user, in this case it is not so sensitive, but is a good practice to maintain JWT tokens when implementing login/signup functionality. Job Recommendation Integration:

We have created a function named as similar-jobs which works to recommend jobs, provides a document of 10 jobs. We then return this to the client side to be presented. Description to the above is given in a different section.

Frontend Details:

The frontend works on a simple create-react-app boiler plate code. Code written in JSX. To implement the multi page site, have used the Route class in react-router-dom package.



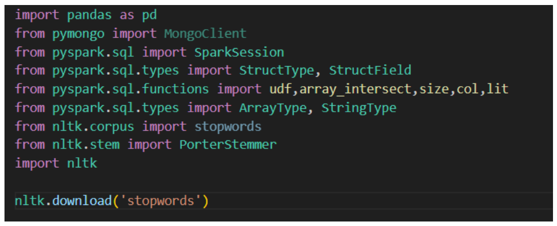
```
@app.route('/api/signup', methods=['POST'])
@app.route('/api/login', methods=['POST'])
@app.route('/decode_jwt', methods=['POST'])
@app.route('/save-job-description', methods=['POST'])
@app.route('/add-skills', methods=['POST'])
@app.route('/recommend-jobs', methods=['POST'])
```

Fig. 1. broad flow of the site

The above flow 1 shows the broad flow in the site.
As is evident, the styling is majorly in manual, flexbox css
The dynamic responsiveness, is attributed to the fetch api builtin js to fetch the server and respond with data of the logged in user, and accordingly interacts the server, and conditionally updates the database as and when needed.

C. TECH STACK BACK-END EXPLANATION

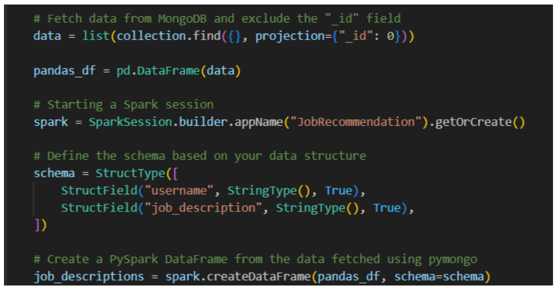
Imports :



```
import pandas as pd
from pymongo import MongoClient
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField
from pyspark.sql.functions import udf,array_intersect,size,col,lit
from pyspark.sql.types import ArrayType, StringType
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
import nltk

nltk.download('stopwords')
```

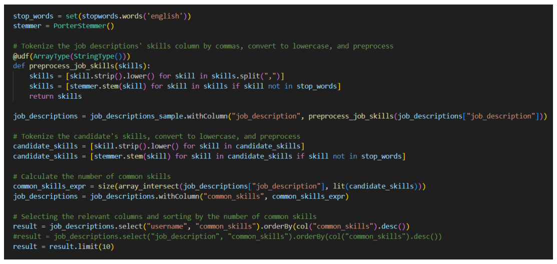
Importing necessary libraries (Stop Words and Porter stemmer from NLTK, Pandas for dataframe handling and necessary modules from Pyspark) for implementing the architecture.

Spark Environment:
Starting a spark session and creating a spark data frame by defining a schema for it.



```python
# Fetch data from MongoDB and exclude the "_id" field
data = list(collection.find({}, projection={"_id": 0}))

pandas_df = pd.DataFrame(data)

# Starting a Spark session
spark = SparkSession.builder.appName("JobRecommendation").getOrCreate()

# Define the schema based on your data structure
schema = StructType([
    StructField("username", StringType(), True),
    StructField("job_description", StringType(), True),
])

# Create a PySpark DataFrame from the data fetched using pymongo
job_descriptions = spark.createDataFrame(pandas_df, schema=schema)
```

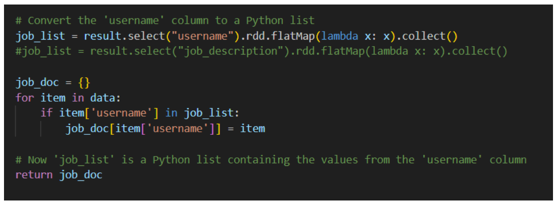
Data Pre-processing:



```python
stop_words = set(stopwords.words('english'))
stemmer = PorterStemmer()

# Tokenize the job descriptions' skills column by commas, convert to lowercase, and preprocess
@udf(ArrayType(StringType()))
def preprocess_job_skills(skills):
    skills = [skill.strip().lower() for skill in skills.split(",")]
    skills = [stemmer.stem(skill) for skill in skills if skill not in stop_words]
    return skills

job_descriptions = job_descriptions_sample.withColumn("job_description", preprocess_job_skills(job_descriptions["job_description"]))

# Tokenize the candidate's skills, convert to lowercase, and preprocess
candidate_skills = [skill.strip().lower() for skill in candidate_skills]
candidate_skills = [stemmer.stem(skill) for skill in candidate_skills if skill not in stop_words]

# Calculate the number of common skills
common_skills_expr = size(array_intersect(job_descriptions["job_description"], lit(candidate_skills)))
job_descriptions = job_descriptions.withColumn("common_skills", common_skills_expr)

# Selecting the relevant columns and sorting by the number of common skills
result = job_descriptions.select("username", "common_skills").orderBy(col("common_skills").desc())
#result = job_descriptions.select("job_description", "common_skills").orderBy(col("common_skills").desc())
result = result.limit(10)
```

Using stop words and porter-stemmer to reduce size of the information and finding the top-10 suitable jobs for a given candidate.

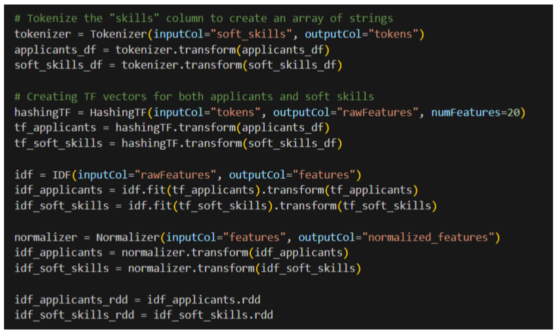
Storing output for applicant- side:



```python
# Convert the 'username' column to a Python list
job_list = result.select("username").rdd.flatMap(lambda x: x).collect()
#job_list = result.select("job_description").rdd.flatMap(lambda x: x).collect()

job_doc = {}
for item in data:
    if item['username'] in job_list:
        job_doc[item['username']] = item

# Now 'job_list' is a Python list containing the values from the 'username' column
return job_doc
```

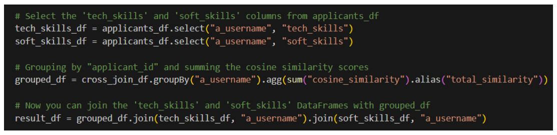
The output is a list of top-10 similar jobs and it is being stored in a dictionary along with skills, soft-skills required for that particular job.

Storing output for hiring side:



```python
# Tokenize the "skills" column to create an array of strings
tokenizer = Tokenizer(inputCol="soft_skills", outputCol="tokens")
applicants_df = tokenizer.transform(applicants_df)
soft_skills_df = tokenizer.transform(soft_skills_df)

# Creating TF vectors for both applicants and soft skills
hashingTF = HashingTF(inputCol="tokens", outputCol="rawFeatures", numFeatures=20)
tf_applicants = hashingTF.transform(applicants_df)
tf_soft_skills = hashingTF.transform(soft_skills_df)

idf = IDF(inputCol="rawFeatures", outputCol="features")
idf_applicants = idf.fit(tf_applicants).transform(tf_applicants)
idf_soft_skills = idf.fit(tf_soft_skills).transform(tf_soft_skills)

normalizer = Normalizer(inputCol="features", outputCol="normalized_features")
idf_applicants = normalizer.transform(idf_applicants)
idf_soft_skills = normalizer.transform(idf_soft_skills)

idf_applicants_rdd = idf_applicants.rdd
idf_soft_skills_rdd = idf_soft_skills.rdd
```

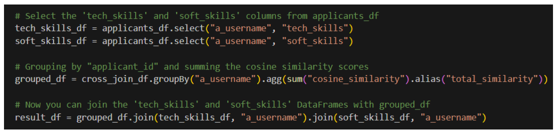
Tokenizing the data as TF-IDF preprocesses text by assigning weights based on their relevance within a document and across a corpus. By doing so, it prepares the data for similarity calculations, enabling more precise comparisons between documents by emphasizing their distinguishing characteristics.



```python
# Select the 'tech_skills' and 'soft_skills' columns from applicants_df
tech_skills_df = applicants_df.select("a_username", "tech_skills")
soft_skills_df = applicants_df.select("a_username", "soft_skills")

# Grouping by "applicant_id" and summing the cosine similarity scores
grouped_df = cross_join_df.groupBy("a_username").agg(sum("cosine_similarity").alias("total_similarity"))

# Now you can join the 'tech_skills' and 'soft_skills' DataFrames with grouped_df
result_df = grouped_df.join(tech_skills_df, "a_username").join(soft_skills_df, "a_username")
```

Code for implementing cosine similarity function to give a similarity score for each applicant according to their skills and skills expecting from the hiring side.

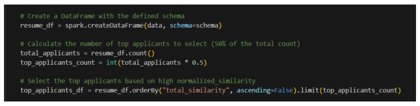
Based on soft-skills of applicants (filtering top 50 percent of the applicants)



```python
# Select the 'tech_skills' and 'soft_skills' columns from applicants_df
tech_skills_df = applicants_df.select("a_username", "tech_skills")
soft_skills_df = applicants_df.select("a_username", "soft_skills")

# Grouping by "applicant_id" and summing the cosine similarity scores
grouped_df = cross_join_df.groupBy("a_username").agg(sum("cosine_similarity").alias("total_similarity"))

# Now you can join the 'tech_skills' and 'soft_skills' DataFrames with grouped_df
result_df = grouped_df.join(tech_skills_df, "a_username").join(soft_skills_df, "a_username")
```

The output contains the dataframe of applicants ordered based on their similarity score with respect to their soft-skills.

Top 50 percent of people are selected for next layer processing based on their similarity score



```python
# Create a DataFrame with the defined schema
resume_df = spark.createDataFrame(data, schema=schema)

# Calculate the number of top applicants to select (50% of the total count)
total_applicants = resume_df.count()
top_applicants_count = int(total_applicants * 0.5)

# Select the top applicants based on high normalized_similarity
top_applicants_df = resume_df.orderBy("total_similarity", ascending=False).limit(top_applicants_count)
```

Thus the new data frame is used to find applicants with required skills using similarity and other functions as discussed previously.

V. MORE ABOUT DATASET USED

Descriptions for each of the columns in the dataset:

Job Id: A unique identifier for each job posting. Experience: The required or preferred years of experience for the job.

Qualifications: The educational qualifications needed for the job.

Salary Range: The range of salaries or compensation offered for the position.

Location: The city or area where the job is located.

Country: The country where the job is located.

Latitude: The latitude coordinate of the job location.

Longitude: The longitude coordinate of the job location. Work Type: The type of employment (e.g., full-time, part-time, contract).

Company Size: The approximate size or scale of the hiring company.

Job Posting Date: The date when the job posting was made public.

Preference: Special preferences or requirements for applicants (e.g., Only Male or Only Female, or Both)

Contact Person: The name of the contact person or recruiter for the job.

Contact: Contact information for job inquiries.

Job Title: The job title or position being advertised.

Role: The role or category of the job (e.g., software developer, marketing manager).

Job Portal: The platform or website where the job was posted.

Job Description: A detailed description of the job responsibilities and requirements.

Benefits: Information about benefits offered with the job (e.g., health insurance, retirement plans).

Skills: The skills or qualifications required for the job.

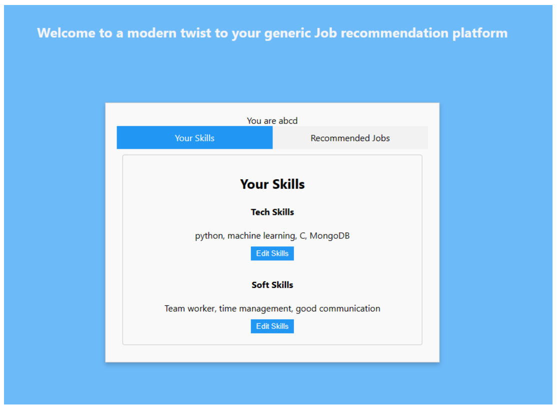
Responsibilities: Specific responsibilities and duties associated with the job.

Company Name: The name of the hiring company.

Company Profile: A brief overview of the company's background and mission.

Taking memory size into consideration, we have used minimum number of columns in our project and this can be extended to using all the columns for more information

VI. RESULTS AND CONCLUSION



Welcome to a modern twist to your generic Job recommendation platform

You are abcd

Your Skills | Recommended Jobs

Your Skills

Tech Skills

python, machine learning, C, MongoDB

Edit Skills

Soft Skills

Team worker, time management, good communication

Edit Skills

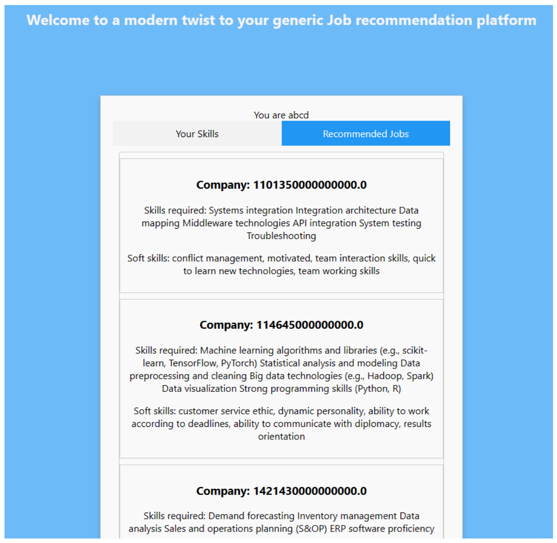
Fig. 2. input

Required information about the applicants is taken as input 2 and top-10 suitable jobs are recommended as shown in 3.

The job recommender system (JRS) is a critical tool in today's recruitment landscape, leveraging various techniques like stop words removal, Porter stemmer, tf-idf text mining, and cosine similarity functions. These methods optimize the system's efficiency by meticulously refining job descriptions and resumes, transforming textual data into structured matrices for robust comparison.

While the current emphasis is on tailoring recommendations primarily for IT roles, the system's potential for diversifying into non-IT sectors presents an exciting prospect for future development.

The literature review on JRS not only delves into technical aspects but also sheds light on broader considerations. It



Welcome to a modern twist to your generic Job recommendation platform

You are abcd

Your Skills | Recommended Jobs

Company: 1101350000000000.0

Skills required: Systems integration Integration architecture Data mapping Middleware technologies API integration System testing Troubleshooting

Soft skills: conflict management, motivated, team interaction skills, quick to learn new technologies, team working skills

Company: 1146450000000000.0

Skills required: Machine learning algorithms and libraries (e.g., scikit-learn, TensorFlow, PyTorch) Statistical analysis and modeling Data preprocessing and cleaning Big data technologies (e.g., Hadoop, Spark) Data visualization Strong programming skills (Python, R)

Soft skills: customer service ethic, dynamic personality, ability to work according to deadlines, ability to communicate with diplomacy, results orientation

Company: 1421430000000000.0

Skills required: Demand forecasting Inventory management Data analysis Sales and operations planning (S&OP) ERP software proficiency

Fig. 3. output

explores the impact of data science competitions, revealing their influence on method selection and system design.

Ethical concerns and algorithmic fairness emerge as crucial factors, highlighting the need for more comprehensive discussions in future JRS research. Additionally, the novel categorization of hybrid recommender systems offers a deeper understanding of their functionality, although there's a persistent need to address interaction data gaps for more effective recommendations.

Recognizing the comprehensive scope of JRS contributions, the review acknowledges inherent limitations. While categorizing hybrid JRS approaches offers more granularity, some overlap between methods persists. Moreover, the review's exclusive focus on JRS may overlook interconnected systems like e-recruitment platforms, prompting further investigation into holistic solutions within the recruitment domain.

The evolving nature of algorithms and data availability accentuates the JRS's role in continually evolving to meet the changing demands of the job market.

This research marks a pivotal moment in JRS analysis, underscoring the need for ongoing system refinement and expansion beyond its current IT-centric focus. The identified limitations provide a roadmap for future improvements, aiming to encompass a broader job spectrum and enhance similarity detection beyond cosine similarity.

As technology advances, exploring alternative similarity measures promises more accurate and diversified job recommendations, aligning with the evolving needs of both candidates and recruiters.

In summary, while the JRS landscape presents a comprehensive understanding of current advancements and challenges, it beckons further exploration. Emphasizing not only technical facets but also ethical considerations and system evolution, the review sets the stage for a more comprehensive and inclusive job recommendation framework that adapts to the dynamic job market landscape.