**MSDS 697 Distributed Data Systems | Group Project**

Laila Zaidi, Param Mehta, Sonal Shad, Shagun Kala, Belinda Ong

# [MSDS697 Group 16 Presentation](https://docs.google.com/presentation/d/1WXKFvlL9Ly7K965DmQaD3suyYKQSNnR1j39DVB9gzic/edit#slide=id.p)

# 

# 

# Executive Summary

This [Github](https://github.com/sonalshad/job-matching-pipeline) repository contains a job matching pipeline using Google Cloud Storage (GCS), Airflow, PySpark, MongoDB, LLM packages on Python, Streamlit, and Google Vision OCR. The pipeline fetches job listing data from the JSearch API from RapidAPI, stores the data in GCS, cleans the data with PySpark, and loads the cleaned data into MongoDB where additional transformations are applied to calculate aggregated statistics and Langchain is used to create embeddings for automated job matching. When resumes are uploaded, Google Vision OCR parses out the relevant content and Langchain is used to transform it into embeddings that can be compared to the records in MongoDB. Based on cosine similarity, the top three jobs are displayed on the Streamlit user interface. We then used an LLM summarizer to synthesize the job descriptions of the top three roles.

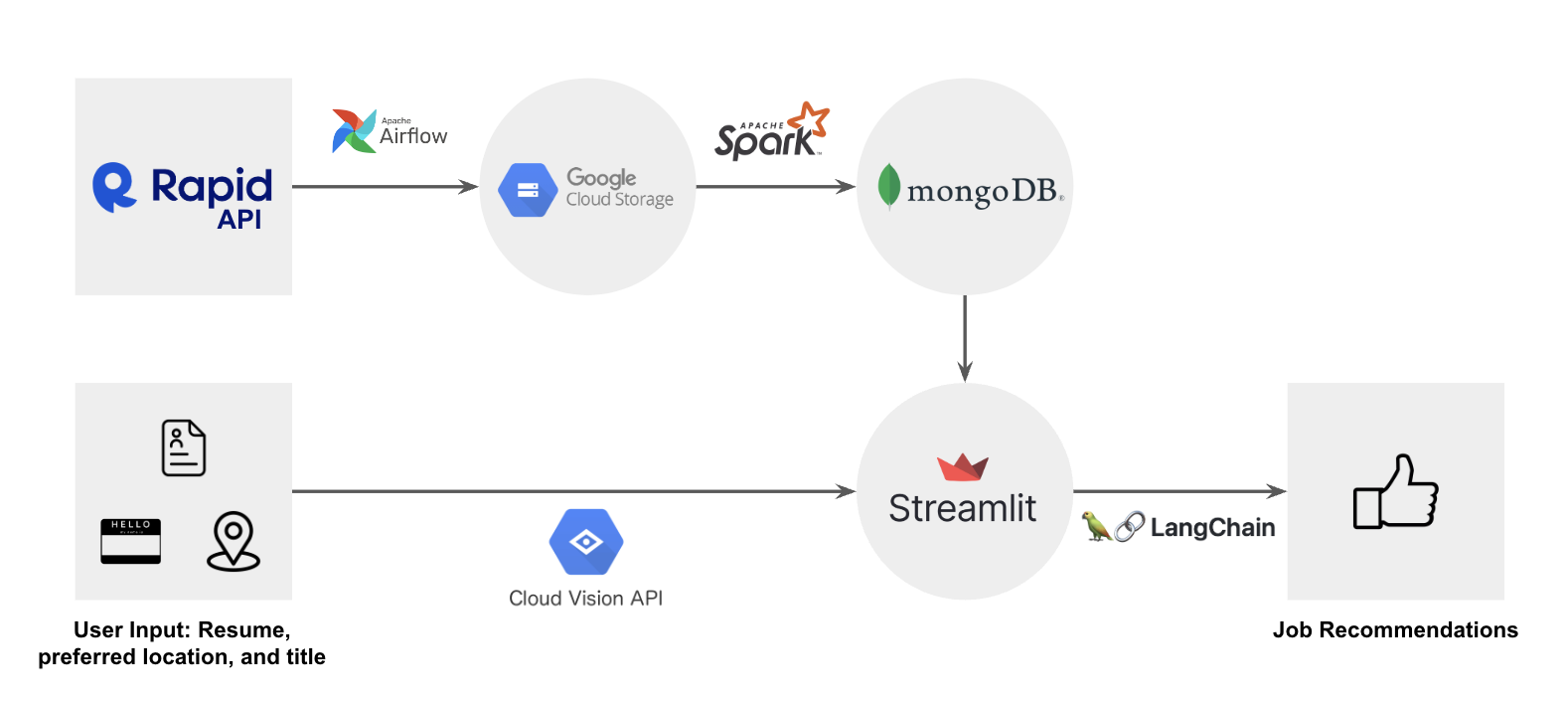
In today’s dynamic job market, finding the perfect role can be a tedious process. Search tools on traditional job aggregation portals (e.g., LinkedIn, Ziprecruiter, and Indeed) are not very useful for leading edge fields such as Data Science, where responsibilities are not uniformly defined across the industry. As a result, job seekers from these disciplines have to spend time searching multiple job titles and reading through individual job descriptions to understand if it is a fit for their profile. Recognizing this challenge, our team looks to make searching for jobs easier by matching a user to open roles based on semantic similarity between their resume and available job postings. We filter jobs based on preferred job titles and location, and match the job descriptions to the user’s resume using an embedding model. For this implementation, we have focused on a limited set of job titles (Data Scientist, Data Analyst, Machine Learning Engineer) and locations (by state: CA, IL, TX).

We built an automated pipeline that routinely, retrieves, stores, and processes data as well as an interface for users to select filters and submit their resume. Submission prompts automated resume-parsing and text processing, followed by similarity ranking. Three jobs with the highest similarity scores are returned to the user via the same interface. In addition, we also implemented an LLM-based algorithm to briefly summarize the three job descriptions.

We sourced our data from theJSearch API using RapidAPI ([Source](https://rapidapi.com/letscrape-6bRBa3QguO5/api/jsearch)): a real-time database that scrapes job postings from LinkedIn, Indeed, Glassdoor, and Google. The data is received as JSON documents with more than 30 possible data points each including job title, location, employer, employment type, category, industry/company type, job requirements, date posted, etc. We chose this data source because it aggregates postings from multiple online platforms, updates in real time, and includes many features, which can be incorporated into future iterations of this project.

**Data Engineering Pipeline**

The JSearch API updates frequently so we utilize Airflow to schedule regular pulls into our automated data pipeline daily. The raw data is in JSON format, which makes MongoDB an ideal DBMS for querying. We utilize dataframes in PySpark to pre-process and aggregate data before storing it as collections in MongoDB. Our model generates recommendations in real-time when triggered by a user event, which requires us to conduct different data wrangling steps (daily aggregations and pre-processing vs. summary of recommendations in real time) via PySpark and MongoDB, as latency differs between the two.



Central to our project is the concept of semantic similarity – the ability to match job postings with user resumes based on the semantic resemblance between them. To effectively filter data, compute statistics, and rank similarity, we need to process the vast amount of data contained within job listings efficiently. Our preprocessing pipeline focuses on extracting and transforming key features such as location, titles, salaries, and job descriptions.

Location and Titles for Filtering: By extracting location and job titles, we enable users to filter job listings based on their preferences, narrowing down their search to the most relevant opportunities. We use MongoDB for efficient querying, filtering, and aggregating.

Statistics on Salaries: Understanding salary ranges within job postings provides valuable insights for job seekers, allowing them to gauge the competitiveness and compensation potential of different roles.

**Optimizing Text Data**

Transforming Job Descriptions

Our Machine Learning model lies in the semantic matching between job descriptions and user resumes. To achieve this, we needed to ensure that the text data in both is informative and conducive to semantic analysis. This involves tokenizing, removing stopwords, and other text preprocessing techniques, all of which are executed using PySpark libraries.

Parsing Resume

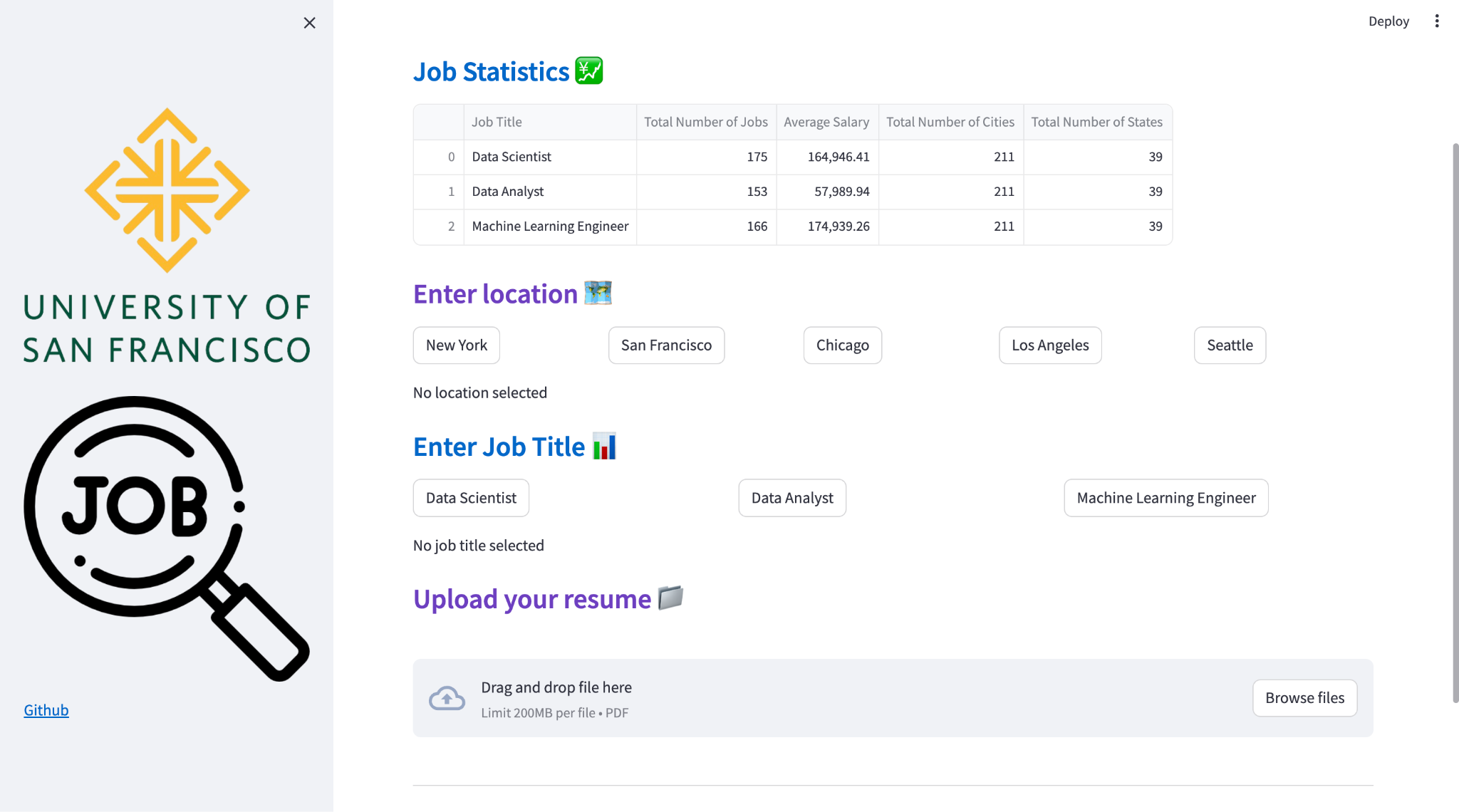
The model accepts user input of the resume as a PDF file, which must be parsed and the text extracted before pre-processing can take place. This is accomplished using Google Vision Pro.

**Similarity Ranking**

Finally, pre-processed text data from the job descriptions is converted into vector embeddings using the HuggingFace Instructor model and stored in MongoDB. When a user uploads their resume, the pre-processed text is converted to an embedding. The model computes cosine similarity between this target embedding and all others in the job data bank and ranks them. Details from the three jobs with the highest similarity are queried from MongoDB to be displayed to the User. In addition, job descriptions for each are accessed and summarized using Gemini.

**User Interface**

Our user interface application is built using the Streamlit library. This application allows users to select location and job title filters, upload their resume with a simple and convenient Graphical User Interface, and receive results in a visually appealing manner. Streamlit enables us to provide a pleasant user experience in addition to the functionality of our model.

****

| **MongoDB:** Due to its in-memory structure, MongoDB queries are faster for simple queries and when limited to basic analytics. Therefore, we used it for initial exploration and straight forward queries such as the total number of jobs or average salary per location.  **Dataframes in Spark:** Due to its parallelization capabilities, Spark scales better and supports advanced analytics and complex aggregations. However, it is slower for simple queries. Therefore, we used it to conduct cleaning and joining functions such as mapping the raw JSON fields to our desired format in the MongoDB collection. Specifically, we used it to concatenate location data from multiple fields such as city, state, and country; create a salary range from the minimum and maximum salary levels identified; remove stopwords, and conduct tokenization and lemmatization. |
| --- |

# Conclusion

Through this job matching pipeline, we successfully reduced the time needed to find roles that align closely with a User’s preferred job title within an up and coming job family: data science. By matching the top roles to given resumes, we demonstrate the seamless integration of various technologies in an automatically updated data pipeline, with data stored across multiple platforms such as GCS and MongoDB clusters. Throughout this endeavor, we navigated decisions regarding data processing, storage, and technology use based on the required speed and efficiency, while balancing resource constraints (e.g., simple queries using MongoDB and parallelized complex data cleaning through Spark Dataframes). This project has been a valuable exploration into practical applications of data engineering, where data from diverse sources necessitates aggregation and processing strategies tailored to different speeds and requirements . Finally, we deployed industry standard leading-edge solutions such as Google Vision OCR, LangChain, Hugging Face, and Gemini to implement a proof-of-concept model with good potential applications and avenues for further growth. In future iterations, the project may be scaled up by increasing the number of monthly API calls (from 200) and corresponding storage. We may also seek to determine the quality of job listings and refine filtering processes as JSearch currently does not provide assurance metrics around scam jobs, broken links, or bot posts. This would improve the reliability and utility of our product.