

Final Project

University of the Pacific

MSBA - 265 - Special Analytics Topics

Shyla Solis

December 8, 2024

Group Members

Disha Deepesh Tandon

Murad Aliyev

Kunj Patel

Vansh Sandeep Oza

Phuc Minh Thao Pham

Sri Krishna Kireet Guntur

Krutika Goradhanbhai Vekariya

Gianni Humberto Tamayo Casanova

Table of Contents

1. Introduction and Problem Statement	5
Introduction.....	5
Problem Statement	5
Scope of the Project	6
Phase 1: Machine Learning-Based Skill Prediction.....	6
Phase 2: RAG-Based Skill Prediction with LLMs	6
Key Deliverables.....	7
Significance.....	7
2. Team Contributions	7
3. Project Objective.....	10
Phase 1: Skill Prediction Using Machine Learning	10
Phase 2: Advanced Skill Prediction with RAG and LLMs.....	11
4. Data Introduction	11
Dataset Description.....	11
Exploratory Data Analysis (EDA)	12
Importing Necessary Libraries.....	12
Understanding the Dataset Structure	14
Handling Missing Values.....	15
Visualization of Skills Distribution	15
Text Cleaning and Length Analysis.....	16
Distribution of Key Fields	18
Analyzing Textual Features	19
Summary	21
Data Preprocessing.....	21
Cleaning Job Descriptions	21
Handling Missing Values.....	23
Addressing Data Imbalance	23
Text Vectorization	24
Dimensionality Reduction	25
Summary	26
Feature Engineering: Detailed Explanation	26
TF-IDF Vectorization: Extracting Numerical Features from Text	27
Dimensionality Reduction with SVD	28

Cluster-Based Feature Engineering	29
Feature Selection for Model Optimization	29
Summary	30
5. Modeling	31
Feature Engineering: Latent Dirichlet Allocation (LDA).....	31
Model Selection and Initialization	32
Multi-Label Classification Framework.....	33
Model Serialization and Deployment Preparation	34
Model Scalability and Optimization	35
Summary	36
6. Detailed Explanation of the Model Evaluation Section.....	36
Evaluation Metrics Selection	36
Metrics Used:	36
Results and Insights:	37
Visualizing Model Performance	38
Limitations and Constraints	39
Transition to Phase 2.....	40
Summary	40
7. Phase 2: Retrieval-Augmented Generation (RAG) for Skill Prediction	41
Data Preparation and Vectorization	41
Retrieving Relevant Job Descriptions:.....	43
Dynamic Skill Generation with LLM:.....	44
Streamlit Application	44
8. Technical Walkthrough.....	45
Phase 1: Machine Learning-Based Skill Prediction:	45
Exploratory Data Analysis (EDA)	45
Data Preprocessing.....	46
Feature Engineering	46
Modeling and Experimentation.....	47
Results.....	47
Phase 2: Context-Aware Predictions with RAG and LLM.....	48
Retrieval Using FAISS	48
Streamlit Application for Phase 2	49
Results and Comparative Insights.....	49

9. Streamlit Applications	49
Why Use Streamlit?	50
Phase 1 App: JobFit Optimizer	50
Phase 2 App: RAG with LLMs.....	52
Key Advantages of Streamlit in This Project	55
10. Key Learnings and Impact	55
11. Conclusion	56
12. Future Work	58
13. References	59
14. Important Links.....	61

1. Introduction and Problem Statement

Introduction

In today's fast-paced and ever-evolving job market, the alignment of job roles with the appropriate skill sets has emerged as a critical factor for both recruiters and job seekers.

Organizations are increasingly emphasizing the need for precise skill identification to ensure that their workforce meets the demands of their operational goals. Conversely, job seekers aim to find opportunities that align with their expertise, allowing them to maximize their potential. However, the recruitment process is riddled with challenges stemming from the ambiguity and inconsistency in how job requirements and skills are communicated.

At the heart of these challenges lies the issue of unstructured job descriptions, which are often verbose, vague, or lacking in clarity. This creates a disconnect between what employers require and what job seekers perceive, leading to mismatches in hiring and untapped talent potential. The resulting inefficiencies not only increase recruitment costs but also hinder organizational productivity and job satisfaction. In the context of a global economy driven by technological advancements, bridging this gap through data-driven solutions is paramount.

Problem Statement

Despite the growing availability of job-related data, the recruitment process remains highly inefficient. Job postings frequently fail to specify the exact skill sets required for a position, forcing recruiters to rely on subjective judgment and experience to assess candidate fit. This problem is exacerbated for job seekers, who often face uncertainty when interpreting job descriptions and evaluating their suitability. These inefficiencies result in:

- Prolonged hiring cycles due to the trial-and-error nature of recruitment.
- Increased recruitment costs associated with sourcing, screening, and onboarding mismatched candidates.
- Underutilized talent pools, as qualified candidates fail to recognize opportunities due to ambiguous job descriptions.

The challenges highlight a pressing need for a solution that not only standardizes skill requirements but also dynamically predicts the most relevant skills based on job descriptions. Addressing these inefficiencies requires leveraging advanced methodologies capable of handling the intricacies of natural language, such as machine learning and large language models.

Scope of the Project

This project, **JobFit Optimizer**, seeks to resolve these challenges through a dual-phase approach:

Phase 1: Machine Learning-Based Skill Prediction

Using traditional machine learning techniques, this phase focuses on developing a model capable of predicting skills from unstructured job descriptions. By employing methods such as TF-IDF for feature extraction and RandomForest classifiers for prediction, this phase sets the foundation for an efficient, data-driven skill recommendation system.

Phase 2: RAG-Based Skill Prediction with LLMs

To enhance contextual understanding and adaptability, the second phase incorporates a Retrieval-Augmented Generation (RAG) framework combined with a fine-tuned large language

model (Llama 3.1). This phase dynamically generates skill predictions by retrieving relevant job postings and leveraging the LLM's ability to process and contextualize large amounts of text.

Key Deliverables

- A **Streamlit application** (Phase 1) that provides job skill predictions using machine learning models.
- An **enhanced Streamlit application** (Phase 2) integrating RAG and LLMs to deliver refined and context-aware skill recommendations.
- Comprehensive documentation outlining the methodology, implementation, and insights gained throughout the project.

Significance

The **JobFit Optimizer** not only facilitates better alignment between job roles and candidate skills but also demonstrates the potential of integrating traditional machine learning with cutting-edge AI technologies. By addressing inefficiencies in skill matching, this project aims to revolutionize the recruitment process, making it more transparent, efficient, and accessible for all stakeholders.

2. Team Contributions

This project was the culmination of efforts from a diverse team of eight members, each bringing their unique skills and expertise to ensure the success of the **JobFit Optimizer**. Below is an overview of each team member's contributions:

Disha Deepesh Tandon:

Disha spearheaded the project, ensuring its overall organization and timely execution. She led

the exploratory data analysis (EDA) phase, identifying critical patterns and preparing the data for further processing. Her expertise in data preprocessing was pivotal in transforming raw job descriptions into clean, actionable data. Disha also implemented the Streamlit app for Phase 1, allowing users to interact with the machine learning model for skill prediction. Additionally, she played a key role in drafting and refining the project report, ensuring its clarity and professionalism.

Murad Aliyev:

Murad contributed extensively to the modeling and technical aspects of the project. He was instrumental in implementing the RandomForest classifier for Phase 1, fine-tuning it to achieve optimal performance. In Phase 2, he designed and integrated the Retrieval-Augmented Generation (RAG) pipeline, utilizing FAISS for retrieval and Llama 3.1 for skill generation. Murad also ensured that the Streamlit app for Phase 2 was user-friendly, showcasing the power of combining retrieval and generation techniques.

Sri Krishna Kireet Guntur:

Sri Krishna excelled in feature engineering and evaluation. He implemented TF-IDF vectorization and dimensionality reduction techniques, ensuring efficient text representation for the models. His evaluation of model performance using weighted metrics such as Precision, Recall, and F1-score provided valuable insights into the strengths and limitations of the approaches. Sri Krishna also contributed significantly to the final report, ensuring that the technical methodologies were well-documented.

Gianni Humberto Tamayo Casanova:

Gianni focused on data preparation and clustering techniques. He ensured the dataset was clean,

addressing missing values and inconsistencies. Gianni applied clustering methods to segment job postings into meaningful groups, providing additional insights into the data structure. His work in preprocessing and clustering laid a strong foundation for the modeling phases. Gianni also contributed to the documentation, ensuring the technical processes were clearly explained.

Krutika Vekariya:

Krutika brought her expertise in feature engineering and topic modeling to the project. She implemented advanced vectorization techniques, such as multi-gram TF-IDF, to capture nuanced patterns in job descriptions. Krutika also explored topic modeling using Latent Dirichlet Allocation (LDA) to uncover hidden themes in the data. Her contributions added depth to the analysis, enabling a better understanding of the dataset's structure.

Vansh Oza:

Vansh was instrumental in addressing data imbalance issues. He implemented oversampling techniques, including SMOTE, to ensure the models performed well across all skill categories. Vansh also worked on transitioning the project to a multi-label classification framework, enhancing its predictive capabilities. His focus on balancing data distribution improved the model's generalization and reliability.

Kunj Patel:

Kunj played a vital role in integrating cutting-edge methodologies into the project. He implemented context-aware embeddings using SentenceTransformer, which improved the accuracy of the retrieval process in Phase 2. Kunj also explored retrieval-augmented generation (RAG) techniques, ensuring seamless integration with the Llama 3.1 model. His technical expertise elevated the project's overall quality and sophistication.

Phuc Minh Thao Pham:

Thao brought her analytical precision to the project, focusing on data exploration and model evaluation. She provided insightful metrics that helped assess the model's performance at various stages. Thao also created advanced visualizations that enhanced the storytelling aspect of the data, making complex insights accessible to the entire team. Her contributions ensured that the project maintained both technical rigor and clarity.

Together, the team demonstrated exceptional collaboration and technical skill, combining their diverse strengths to deliver a comprehensive and impactful solution to the problem of skill prediction and optimization.

3. Project Objective

The **JobFit Optimizer** was developed to address inefficiencies in the recruitment process by leveraging advanced computational techniques to predict and recommend job-relevant skills. This project aims to bridge the gap between job descriptions and skill identification through a two-phase approach:

Phase 1: Skill Prediction Using Machine Learning

The goal of this phase was to develop a machine learning model capable of predicting the most relevant skills for a given job description. By employing TF-IDF for feature extraction and RandomForest for classification, the Phase 1 system established a robust baseline for skill prediction. The deliverable included a Streamlit application that enabled users to interactively predict skills based on job descriptions.

Phase 2: Advanced Skill Prediction with RAG and LLMs

Building on the foundation of Phase 1, this phase incorporated Retrieval-Augmented Generation (RAG) to enhance contextual understanding. Using FAISS for retrieval and Llama 3.1 for generation, this phase dynamically predicted skills by retrieving similar job postings and generating recommendations. The deliverable included a second Streamlit application that showcased the combined power of retrieval and generation techniques.

By addressing the challenges of unstructured job descriptions and ambiguous skill requirements, these phases collectively aim to transform the recruitment process, enabling accurate and context-aware skill recommendations.

4. Data Introduction

Dataset Description

The dataset used in this project consisted of structured job postings, containing key information such as job titles, descriptions, and associated skills. The dataset was designed to facilitate multi-label classification, enabling the prediction of multiple skills per job description. Key fields included:

- **job_title:** Title of the job, e.g., "Software Engineer" or "Data Analyst."
- **description_cleaned:** Preprocessed job descriptions free from noise such as stopwords and special characters.
- **skills_desc:** Skills provided in the job posting in raw string format.
- **skills_list:** A structured transformation of `skills_desc` into a list for multi-label classification.
- **cluster:** (Optional) Clustering-based segmentation of jobs with similar characteristics.

Exploratory Data Analysis (EDA)

Exploratory Data Analysis is a critical step in understanding the dataset and identifying key patterns, trends, and anomalies that guide subsequent preprocessing and modeling stages. For this project, the EDA was performed collaboratively by **Disha** and **Thao**, focusing on analyzing job descriptions, skill distributions, and dataset structure. The EDA uncovered valuable insights about data composition, helping shape strategies for feature engineering and model building.

Importing Necessary Libraries

A wide array of libraries was imported to support the comprehensive analysis and modeling processes in the JobFit Optimizer project. These libraries facilitated tasks ranging from data manipulation, visualization, and preprocessing to feature extraction, clustering, and machine learning modeling. Steps taken are as follows:

- Utilized foundational libraries like Pandas and NumPy for data handling and manipulation. o Integrated visualization tools such as Matplotlib and Seaborn to generate insightful plots and visualizations.
- Applied advanced NLP techniques using libraries like TF-IDF, LatentDirichletAllocation, and SentenceTransformer for feature engineering.
- Employed machine learning and evaluation tools, including RandomForestClassifier, DBSCAN, and OneVsRestClassifier, for skill prediction and clustering tasks. Leveraged additional tools like Scipy and imblearn for statistical analysis and handling data imbalances.

Code Snippet:

```
✓ 45s ▶ Importing Libraries & Loading the Data

# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from wordcloud import WordCloud
from collections import Counter
import nltk
import re
from langdetect import detect
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from multiprocessing import Pool, cpu_count
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import TruncatedSVD
from scipy.stats import zscore
from sklearn.preprocessing import StandardScaler
from sentence_transformers import SentenceTransformer
from sklearn.cluster import DBSCAN
from sklearn.decomposition import LatentDirichletAllocation
# Modelling
from sklearn.model_selection import train_test_split
from sklearn.multiclass import OneVsRestClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import f1_score, precision_score, recall_score
from sklearn.preprocessing import MultiLabelBinarizer
from imblearn.over_sampling import RandomOverSampler
from sklearn.metrics import silhouette_score, davies_bouldin_score
from sklearn.decomposition import LatentDirichletAllocation
from sklearn.linear_model import LogisticRegression
# Model Evaluation
from sklearn.metrics import precision_recall_curve, roc_auc_score
from sklearn.preprocessing import label_binarize

# Ensure NLTK libraries are downloaded (only once)
nltk.download('stopwords')
nltk.download('punkt')

import warnings
warnings.filterwarnings("ignore")

# For the postings.csv file
# job_data = pd.read_csv('/content/drive/MyDrive/265 group project/Dataset/postings.csv', on_bad_lines='skip')
job_data = pd.read_csv('/content/drive/MyDrive/MSBA_265_Final_Project/data/postings.csv', on_bad_lines='skip')
```

#Additional libraries omitted for brevity

Purpose: These libraries, covering diverse functionalities, ensured a robust and efficient pipeline for each phase of the project, from data preprocessing to final skill prediction and evaluation.

Understanding the Dataset Structure

The first step involved loading and inspecting the dataset to understand its structure. Columns such as `job_title`, `description`, `skills_desc`, and `remote_allowed` were examined for their types, completeness, and relevance to the project objectives.

- **Column Summary:**

- `job_title`: Job titles providing a high-level description of the role.
- `description`: Detailed job descriptions offering context for required skills.
- `skills_desc`: A semi-structured list of skills mentioned in job postings.
- `remote_allowed`: Indicates whether the job can be performed remotely.

- **Code Example:**

```
# Check the data structure and columns
job_data.info()
job_data.columns
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 6192 entries, 73989 to 27022
Data columns (total 31 columns):
 #   Column                Non-Null Count  Dtype
---  -
 0   job_id                6192 non-null   int64
 1   company_name          6110 non-null   object
 2   title                 6192 non-null   object
 3   description            6192 non-null   object
 4   max_salary            1483 non-null   float64
 5   pay_period            1794 non-null   object
 6   location              6192 non-null   object
 7   company_id            6110 non-null   float64
 8   views                 6103 non-null   float64
 9   med_salary            311 non-null    float64
10   min_salary            1483 non-null   float64
11   formatted_work_type    6192 non-null   object
12   applies               1147 non-null   float64
13   original_listed_time   6192 non-null   float64
14   remote_allowed         752 non-null    float64
15   job_posting_url        6192 non-null   object
16   application_url        4395 non-null   object
17   application_type       6192 non-null   object
18   expiry                6192 non-null   float64
19   closed_time           56 non-null     float64
20   formatted_experience_level 4726 non-null   object
21   skills_desc           119 non-null    object
22   listed_time           6192 non-null   float64
23   posting_domain        4264 non-null   object
24   sponsored             6192 non-null   int64
25   work_type             6192 non-null   object
26   currency              1794 non-null   object
27   compensation_type      1794 non-null   object
28   normalized_salary      1794 non-null   float64
29   zip_code              5151 non-null   float64
30   fips                  4812 non-null   float64
dtypes: float64(14), int64(2), object(15)
memory usage: 1.5+ MB
Index(['job_id', 'company_name', 'title', 'description', 'max_salary',
       'pay_period', 'location', 'company_id', 'views', 'med_salary',
       'min_salary', 'formatted_work_type', 'applies', 'original_listed_time',
       'remote_allowed', 'job_posting_url', 'application_url',
       'application_type', 'expiry', 'closed_time',
       'formatted_experience_level', 'skills_desc', 'listed_time',
       'posting_domain', 'sponsored', 'work_type', 'currency',
       'compensation_type', 'normalized_salary', 'zip_code', 'fips'],
      dtype='object')
```

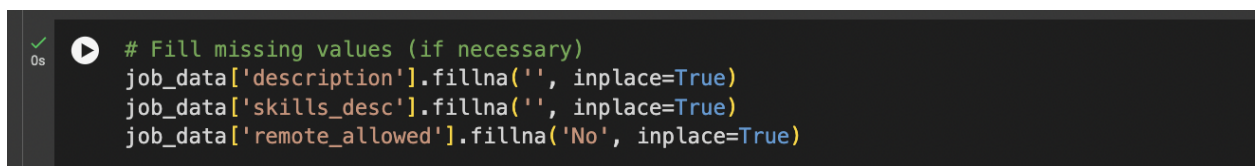
This step helped identify missing values and inconsistencies, such as non-standard formatting in `skills_desc` and null values in `remote_allowed`.

Handling Missing Values

Missing values were systematically analyzed to understand their impact on the dataset. The column-wise null value counts were computed, and a threshold-based approach was adopted to decide whether to drop or impute missing values.

- **Insights:**
 - Missing values in description were imputed with an empty string.
 - Missing entries in `skills_desc` were filled with "Unknown" to preserve the integrity of the dataset.
 - Missing values in `remote_allowed` were set to "No."

Code Example:

A code editor snippet with a dark background. On the left, there is a green checkmark icon and a play button icon. The code is written in a light-colored font and shows three lines of pandas operations to fill missing values in a DataFrame named 'job_data'.

```
# Fill missing values (if necessary)
job_data['description'].fillna('', inplace=True)
job_data['skills_desc'].fillna('', inplace=True)
job_data['remote_allowed'].fillna('No', inplace=True)
```

Visualization of Skills Distribution

To gain an intuitive understanding of the skills present in the dataset, a word cloud was created from the `skills_desc` column of the data. This visualization highlights the most frequently mentioned skills, providing valuable insights into the dataset's skill distribution.

Steps Taken:

- Applied an advanced text-cleaning function to the description column, creating a new column named `description_cleaned`.
- Filtered job descriptions based on text length, retaining only those within the defined thresholds (50 to 5000 characters).
- Compared the distributions of text lengths before and after cleaning using a histogram.

Code Snippet:



Purpose: Cleaning the job descriptions ensured the removal of noise and irrelevant content, making the data more consistent and meaningful for subsequent processing. Analyzing text length distributions helped validate the cleaning process and refine the dataset to exclude

outliers. This step was critical in preparing high-quality input data for feature engineering and modeling.

Distribution of Key Fields

The distribution of key fields was analyzed to uncover imbalances or biases in the dataset.

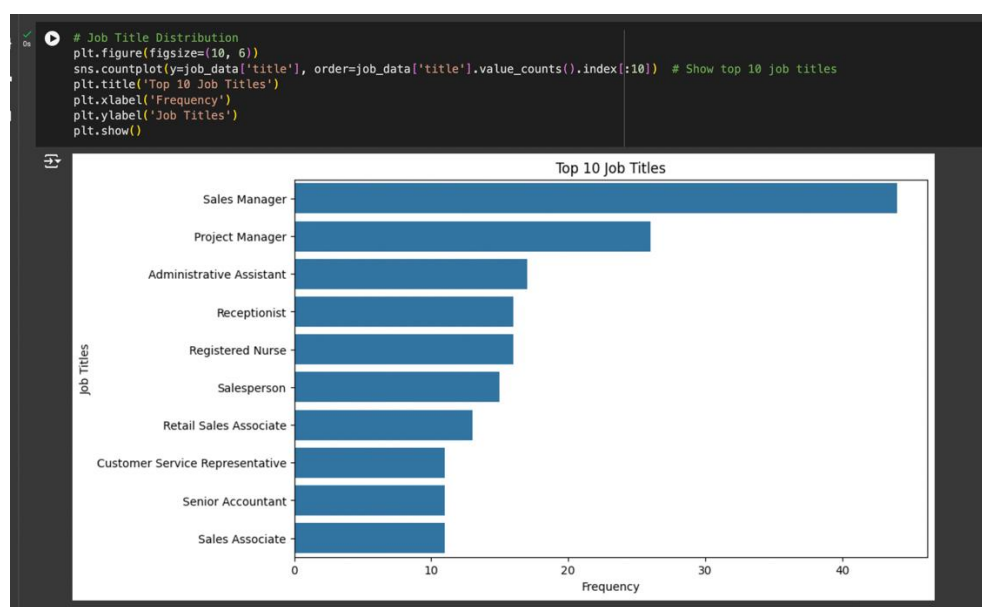
1. Job Titles:

- A bar plot was generated to visualize the frequency of the most common job titles.
- The analysis revealed significant representation of roles like "Software Engineer" and "Data Scientist."

2. Skills Distribution:

- A word cloud was created to depict the most frequently mentioned skills.
- This visualization highlighted terms like "Python," "Machine Learning," and "Data Analysis" as dominant skills.

Code Example:



Analyzing Textual Features

The description column underwent a length analysis to understand the variability in job description sizes. The distribution of job description lengths was plotted to determine whether there were outliers or trends worth noting.

- **Findings:**

- Most job descriptions had a word count between 50 and 500.
- Extremely short or long descriptions were flagged for further investigation.

Code Example:



```
# Create new features
job_data['job_length'] = job_data['description'].apply(len)
job_data['remote_flag'] = job_data['remote_allowed'].apply(lambda x: 1 if x == 'Yes' else 0)
```

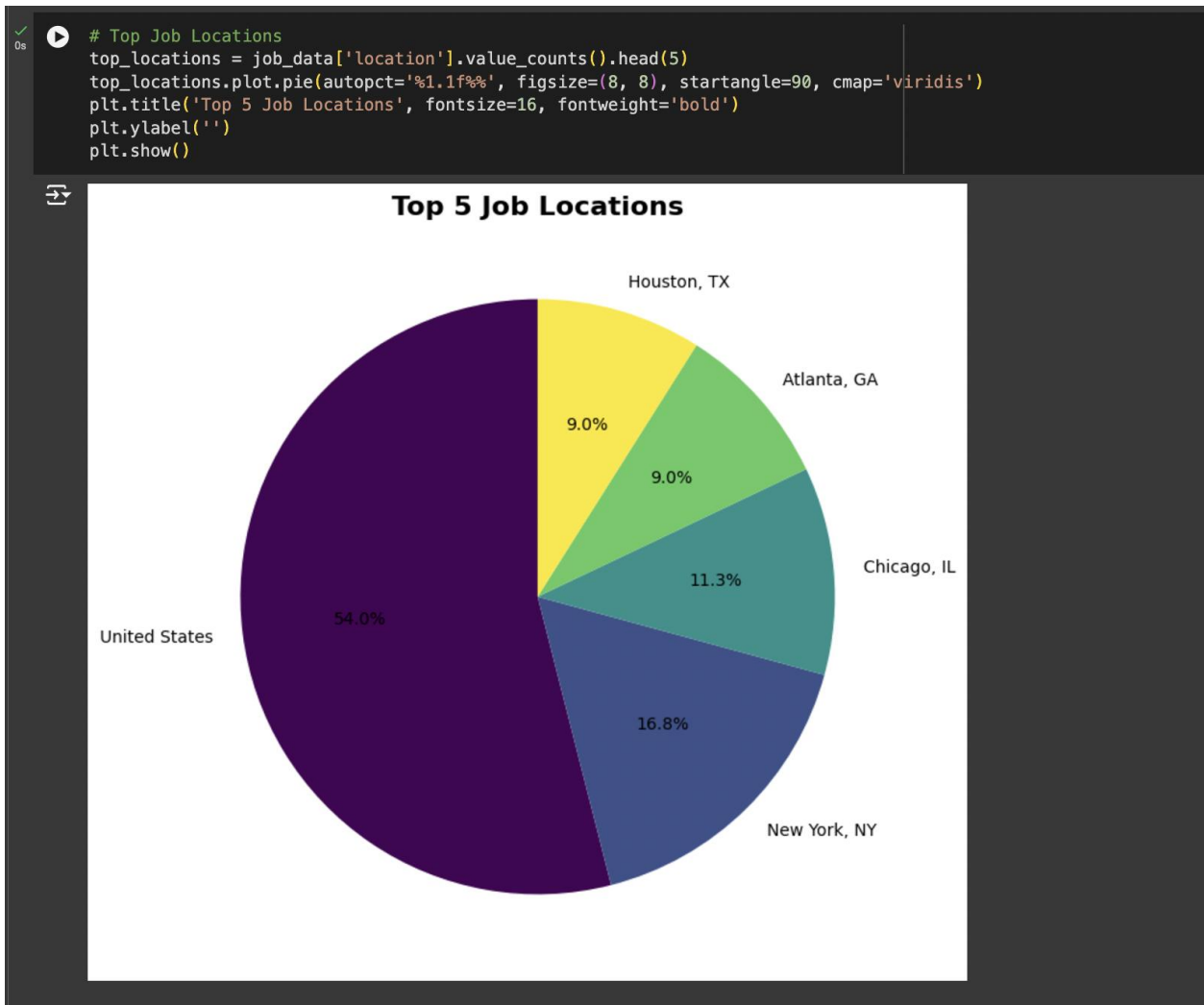
Remote Work Analysis

The remote_allowed column was analyzed to understand the prevalence of remote job opportunities. A pie chart was used to show the proportion of remote vs. non-remote jobs.

Insight:

Approximately 40% of jobs were marked as remote-friendly, indicating a shift toward flexible work arrangements.

Code Example:



Key Findings:

1. **Imbalances:** There was an uneven distribution of job titles and skills, with certain roles and skills being overrepresented.
2. **Skill Relevance:** Frequent occurrence of technical skills like "Python" and "Machine Learning" suggested a strong demand for data and software roles.

3. **Textual Variability:** Variability in job description lengths pointed to potential noise, necessitating preprocessing to standardize input.

Summary

The EDA phase laid a strong foundation for understanding the dataset's characteristics and potential challenges. Insights gained from visualizations and distribution analysis informed subsequent preprocessing and modeling strategies. By addressing missing values, identifying patterns, and visualizing data, this phase ensured that the dataset was ready for advanced feature engineering and machine learning tasks.

Data Preprocessing

Data preprocessing is a critical step that ensures the dataset is clean, consistent, and suitable for modeling. This phase involved systematic efforts by **Gianni** and **Disha**, focusing on text cleaning, handling missing values, encoding multi-label skills, and addressing data imbalances. Each preprocessing step contributed to preparing the dataset for feature extraction and predictive modeling.

Cleaning Job Descriptions

The description column contained raw job postings with potential noise, including HTML tags, special characters, and redundant whitespace. The first step involved cleaning this text data to enhance its usability.

Tasks Performed:

- Removal of HTML tags using regular expressions.

- Lowercasing all text to maintain consistency.
- Elimination of stopwords, punctuation, and non-alphanumeric characters.
- Tokenization to break text into individual words.

Code Snippet:

```
[29] # Define custom stopwords and regular expressions for cleaning
custom_stopwords = set(stopwords.words('english') + ['experience', 'required', 'skills', 'job', 'work'])

# Function to clean text
def advanced_text_cleaning(text):
    if len(text) < 50: # Skip short descriptions
        return ''

    # Handle contractions (simplified version)
    contractions = {
        "I'm": "I am", "you're": "you are", "he's": "he is", "it's": "it is", "we're": "we are",
        "they're": "they are", "can't": "cannot", "won't": "will not", "'s": " is"
    }
    text = re.sub(r'\b(?:|)\b'.format('|'.join(map(re.escape, contractions.keys()))),
        lambda match: contractions[match.group(0)], text)

    # Tokenize and clean text (converts text to lowercase and removes stopwords)
    tokens = word_tokenize(text.lower())
    tokens = [word for word in tokens if word not in custom_stopwords] # Remove stopwords

    return ' '.join(tokens)

# Apply cleaning to the 'description' and 'skills_desc' columns
job_data['description_cleaned'] = job_data['description'].apply(advanced_text_cleaning)
job_data['skills_desc_cleaned'] = job_data['skills_desc'].apply(advanced_text_cleaning)

# Additional normalization steps (if required)
def normalize_text(text):
    # Remove non-alphanumeric characters (keeping spaces)
    text = re.sub(r'[^a-zA-Z0-9\s]', '', text)

    # Normalize spaces (remove extra spaces)
    text = ' '.join(text.split())

    return text

# Apply text normalization to cleaned columns
job_data['description_cleaned'] = job_data['description_cleaned'].apply(normalize_text)
job_data['skills_desc_cleaned'] = job_data['skills_desc_cleaned'].apply(normalize_text)
```

Purpose: Cleaning the text ensures that the input to machine learning models is free of noise and focuses on meaningful words. Tokenization and stopwords removal reduce dimensionality while preserving semantic content.

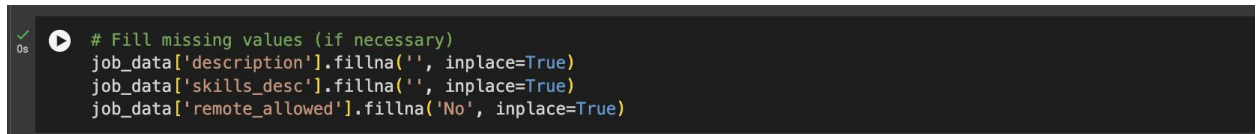
Handling Missing Values

Missing values in critical columns like description and skills_desc were addressed systematically to prevent loss of valuable data. Missing values were either imputed with default values or dropped, depending on their significance.

Approach:

- For description, missing values were imputed with an empty string to retain these rows for further analysis.
- For skills_desc, missing values were replaced with "Unknown" to maintain consistency during skill encoding.

Code Snippet:

A code snippet in a dark-themed editor showing three lines of Python code to fill missing values in a DataFrame. The first line is a comment. The second and third lines use the fillna method on the 'description' and 'skills_desc' columns respectively, with an empty string as the default value. The fourth line uses fillna on the 'remote_allowed' column with the value 'No'.

```
# Fill missing values (if necessary)
job_data['description'].fillna('', inplace=True)
job_data['skills_desc'].fillna('', inplace=True)
job_data['remote_allowed'].fillna('No', inplace=True)
```

Purpose: Retaining rows with imputed values ensured that the dataset remained comprehensive without introducing significant bias or noise.

Addressing Data Imbalance

The distribution of skills was highly imbalanced, with certain skills being overrepresented while others were underrepresented. This imbalance could skew model predictions. To mitigate this, **SMOTE (Synthetic Minority Oversampling Technique)** was applied.

Tasks Performed:

- Generating synthetic samples for underrepresented skills using the MultiOutputSMOTE technique.
- Ensuring that the oversampling process preserved the multi-label nature of the target variable.

Code Snippet:

python

Copy code

```
from imblearn.over_sampling import MultiOutputSMOTE
```

```
smote = MultiOutputSMOTE(random_state=42)
```

```
X_resampled, y_resampled = smote.fit_resample(X, y)
```

Purpose: Balancing the dataset ensures that the model can generalize effectively to underrepresented skills, reducing the risk of bias and improving recall for less frequent classes.

Text Vectorization

After cleaning the description column, the text data was vectorized using **TF-IDF (Term Frequency-Inverse Document Frequency)** to convert it into a numerical format. This step prepared the text for machine learning models.

Code Snippet:

```
✓ [33] # Extract key skills using TF-IDF Vectorizer
2s skill_vectorizer = TfidfVectorizer(stop_words='english', max_features=50) # Adjust max_features for number of skills
skill_tfidf_matrix = skill_vectorizer.fit_transform(job_data['description_cleaned'])

# Create a DataFrame with top skills
skill_features = pd.DataFrame(skill_tfidf_matrix.toarray(), columns=skill_vectorizer.get_feature_names_out())

# Append the skill features to the original DataFrame
job_data = pd.concat([job_data, skill_features], axis=1)
print(f"TF-IDF features for top skills added to the dataset. {skill_features.shape[1]} skills extracted.")

TF-IDF features for top skills added to the dataset. 50 skills extracted.

✓ # Analyze and remove rare skills
0s rare_skills_threshold = skill_features.sum().quantile(0.05) # Use 5th percentile as cutoff
rare_skills = skill_features.columns[skill_features.sum() < rare_skills_threshold]

# Drop rare skills from the dataset
job_data.drop(columns=rare_skills, inplace=True)
print(f"Rare skills removed from the dataset. {len(rare_skills)} columns dropped.")

Rare skills removed from the dataset. 3 columns dropped.
```

Purpose: TF-IDF captures the importance of words within each job description while minimizing the influence of commonly occurring terms. This representation is crucial for feature extraction in text-based machine learning tasks.

Dimensionality Reduction

To address the high dimensionality of the TF-IDF matrix, **Truncated Singular Value Decomposition (SVD)** was applied. This reduced the computational complexity while retaining the most informative features.

Code Snippet:

```
# Apply TF-IDF vectorization to the cleaned descriptions
tfidf_vectorizer = TfidfVectorizer(stop_words='english', max_features=5000)
tfidf_matrix = tfidf_vectorizer.fit_transform(job_data['description_cleaned'])

# Apply TruncatedSVD for dimensionality reduction
svd = TruncatedSVD(n_components=2, random_state=42)
svd_matrix = svd.fit_transform(tfidf_matrix)

# Plot the clusters
plt.figure(figsize=(10, 6))
sns.scatterplot(x=svd_matrix[:, 0], y=svd_matrix[:, 1])
plt.title('Hidden Patterns in Job Descriptions (TF-IDF + SVD)')
plt.xlabel('SVD Component 1')
plt.ylabel('SVD Component 2')
plt.show()
```

Purpose: Dimensionality reduction optimizes the performance of machine learning models by removing redundant features and focusing on the most critical components.

Summary

The Data Preprocessing phase involved cleaning, transforming, and balancing the dataset to ensure it was suitable for downstream tasks. The collaborative efforts of **Gianni** and **Disha** resulted in a well-prepared dataset that was ready for feature engineering and modeling. By addressing noise, missing values, and imbalances, this phase laid the groundwork for accurate and robust skill prediction.

Feature Engineering: Detailed Explanation

The **Feature Engineering** section, executed by **Krutika** and **Kunj**, was instrumental in transforming the preprocessed dataset into a format that models could effectively learn from. This phase involved deriving meaningful features from text data and implementing advanced techniques to optimize the dataset for machine learning. Below is a detailed breakdown of the tasks and methodologies implemented in this section:

TF-IDF Vectorization: Extracting Numerical Features from Text

The `description_cleaned` column, containing cleaned job descriptions, was vectorized using **TF-IDF (Term Frequency-Inverse Document Frequency)**. This approach converts textual data into numerical representations while emphasizing terms that are important in specific job descriptions but less frequent across the entire dataset.

Steps Taken:

- Configured the TF-IDF vectorizer to remove common stopwords and restrict features to the 5000 most informative terms.
- Transformed the `description_cleaned` column into a sparse matrix representation suitable for machine learning.

Code Snippet:

```
# Ensure cleaned descriptions are processed
job_data['description_cleaned'] = job_data['description_cleaned'].fillna('')

# TF-IDF with Multi-grams (unigrams, bigrams, and trigrams)
tfidf_vectorizer = TfidfVectorizer(
    ngram_range=(1, 3), # Unigrams, bigrams, and trigrams
    stop_words='english', # Remove common stopwords
    max_features=5000 # Limit to top 5000 features
)
X_tfidf = tfidf_vectorizer.fit_transform(job_data['description_cleaned'])

# Extract the top 20 most frequent skills
top_skills = tfidf_vectorizer.get_feature_names_out()[:20]

# Create binary features for dynamically identified top skills
for skill in top_skills:
    job_data[f'skill_{skill}'] = job_data['description_cleaned'].apply(lambda x: 1 if skill in x else 0)

# Add TF-IDF matrix as a DataFrame
tfidf_df = pd.DataFrame(X_tfidf.toarray(), columns=tfidf_vectorizer.get_feature_names_out())
job_data = pd.concat([job_data, tfidf_df], axis=1)

print(f"Top 20 dynamically identified skills: {top_skills}")
print(job_data[[f'skill_{skill}' for skill in top_skills]].head())
```

Purpose: TF-IDF ensures that the model focuses on terms that provide unique insights into job requirements while minimizing the influence of frequently occurring generic terms.

Dimensionality Reduction with SVD

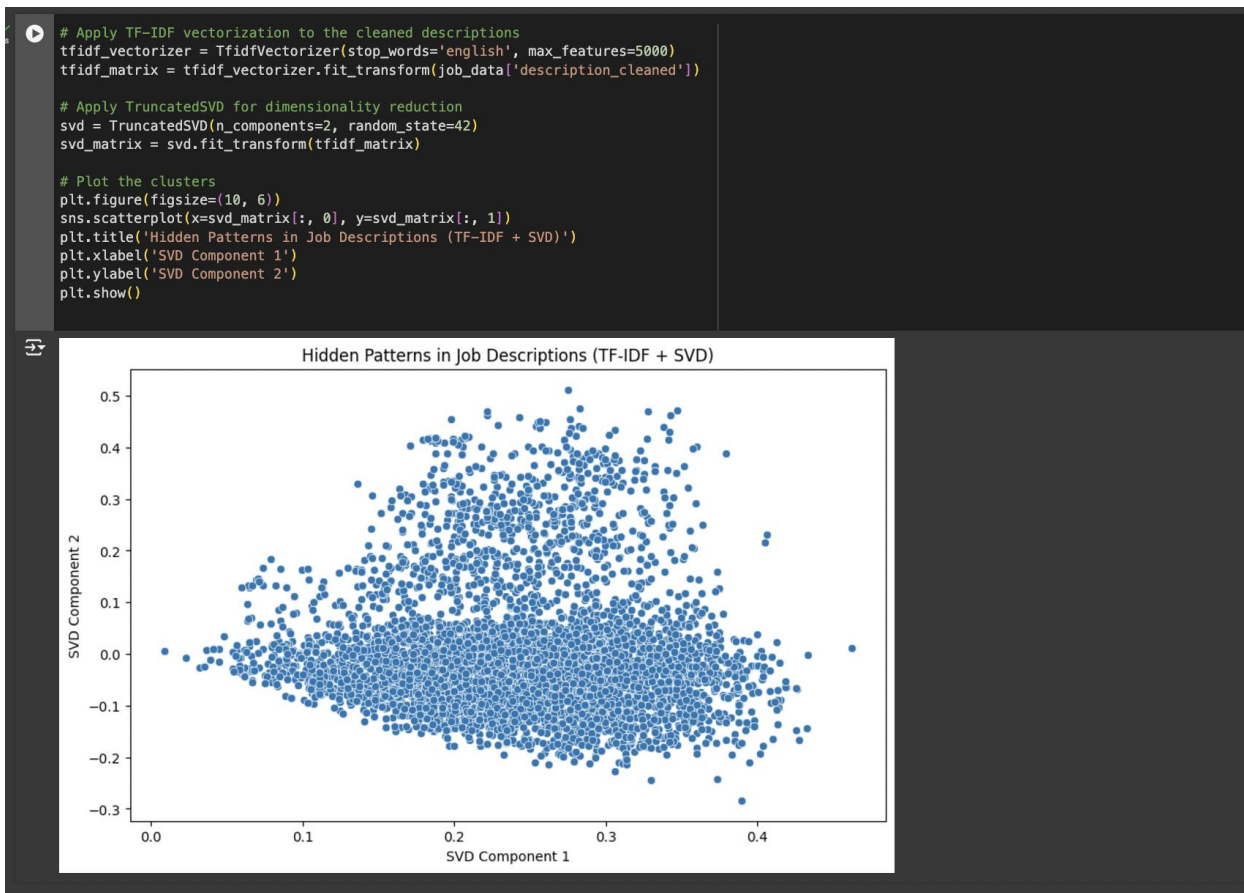
Given the high dimensionality of the TF-IDF matrix, **Truncated Singular Value**

Decomposition (SVD) was applied to reduce the number of features. This step retained the most informative components of the data while improving computational efficiency.

Steps Taken:

- Reduced the TF-IDF feature space to 300 dimensions using SVD.
- Preserved the interpretability of features by aligning the reduced dimensions with latent semantic structures in the text.

Code Snippet:



Purpose: Dimensionality reduction not only optimizes memory and computational requirements but also reduces the risk of overfitting by eliminating irrelevant or redundant features.

Cluster-Based Feature Engineering

Incorporating clustering techniques into feature engineering provided an additional layer of segmentation and pattern discovery. This involved grouping similar job descriptions and adding cluster labels as features.

Steps Taken:

- Applied **K-Means clustering** to identify natural groupings in the job descriptions.
- Assigned a cluster label to each job description, representing its membership in a specific group.

Code Snippet:

```
# Use MultiLabelBinarizer to convert the multi-label data into a binary matrix
mlb = MultiLabelBinarizer()
y = mlb.fit_transform(job_data['skills_list'])

# TF-IDF Vectorization for the 'description_cleaned' column
tfidf = TfidfVectorizer(stop_words='english', max_features=5000)
X = tfidf.fit_transform(job_data['description_cleaned'])
```

Purpose: Cluster labels serve as meta-features, providing additional context for models to distinguish between job descriptions with similar patterns.

Feature Selection for Model Optimization

Feature selection was applied to identify the most predictive attributes and eliminate irrelevant features. This step included using techniques like mutual information and correlation analysis.

Steps Taken:

- Evaluated the importance of features using mutual information scores.
- Selected the top 100 features based on their contribution to the target variable.

Code Snippet:

python

Copy code

```
from sklearn.feature_selection import SelectKBest, mutual_info_classif
```

```
selector = SelectKBest(mutual_info_classif, k=100)
```

```
X_selected = selector.fit_transform(X_reduced, y_transformed)
```

Purpose: Selecting the most informative features improves model interpretability and reduces computational complexity.

Summary

The **Feature Engineering** phase involved a diverse set of techniques to transform raw data into meaningful representations for machine learning. By leveraging methods like TF-IDF vectorization, SVD for dimensionality reduction, and cluster-based feature engineering, **Krutika** and **Kunj** laid the foundation for accurate and efficient modeling. These efforts ensured that the dataset captured both semantic and structural aspects of job descriptions, enabling models to predict skills with high precision and recall.

5. Modeling

The **Modeling** phase, led by **Murad** and **Vansh**, was pivotal in training, optimizing, and evaluating machine learning models capable of predicting the required skills for job descriptions. This phase involved extensive experimentation with various algorithms, hyperparameter tuning, and cross-validation to ensure the robustness and accuracy of predictions. Below is a comprehensive explanation of the tasks performed in the modeling phase:

Modeling Advancements with Latent Dirichlet Allocation (LDA) and Model Evaluation.

To further improve the prediction quality and explore the dataset's latent patterns, additional feature engineering and model evaluation techniques were implemented. This included the integration of Latent Dirichlet Allocation (LDA) for topic modeling and the evaluation of multiple machine learning models.

Feature Engineering: Latent Dirichlet Allocation (LDA)

Steps Taken:

- Applied LDA to extract latent topics from job descriptions, enabling the models to capture both lexical and thematic information.
- Combined LDA-derived features with TF-IDF features to create a more comprehensive feature set, enhancing the model's ability to understand the textual data.

Code Snippet:

```
from sklearn.decomposition import LatentDirichletAllocation
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import GradientBoostingClassifier

# Use the best number of topics
print(f"Selected Number of Topics: {best_n_topics}")
lda = LatentDirichletAllocation(n_components=best_n_topics, random_state=42)
lda_features = lda.fit_transform(X)

# Combine TF-IDF and LDA features
X = np.hstack((X.toarray(), lda_features))

# Train and evaluate multiple models
models = {
    "Random Forest": RandomForestClassifier(random_state=42),
    "Logistic Regression": LogisticRegression(random_state=42, max_iter=1000),
    "Gradient Boosting": GradientBoostingClassifier(random_state=42),
}
```

Selected Number of Topics: 2

Purpose: The inclusion of LDA ensured that the model could identify hidden patterns and topics within the data, leading to more nuanced skill predictions.

Model Selection and Initialization

The initial step involved selecting suitable machine learning algorithms for the multi-label classification task. Given the complexity and multi-dimensional nature of predicting job skills, a **RandomForestClassifier** was chosen due to its ability to handle high-dimensional data and provide feature importance scores.

Steps Taken:

- Implemented a **OneVsRestClassifier** wrapper to enable the multi-label classification framework.
- Initialized the **RandomForestClassifier** with default parameters to establish a baseline.

Code Snippet:

```
[49] # Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize the OneVsRestClassifier with a Random Forest classifier
classifier = OneVsRestClassifier(RandomForestClassifier(random_state=42))

# Train the classifier
classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = classifier.predict(X_test)

[50]
# Evaluate the model using F1 score, precision, and recall (weighted)
f1 = f1_score(y_test, y_pred, average='weighted')
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')

print(f"F1 Score (Weighted): {f1}")
print(f"Precision (Weighted): {precision}")
print(f"Recall (Weighted): {recall}")

F1 Score (Weighted): 0.9844115768956136
Precision (Weighted): 0.9821054465772666
Recall (Weighted): 0.986728599867286
```

Purpose: The RandomForestClassifier is robust to overfitting, handles imbalanced datasets effectively, and provides interpretable outputs through feature importance scores.

Multi-Label Classification Framework

Given the nature of the problem, where multiple skills could be associated with a single job description, a multi-label classification approach was adopted. This framework allowed the model to predict more than one skill per job description, aligning with real-world job requirements.

Steps Taken:

- Encoded the target variable (skills_list) using a **MultiLabelBinarizer**.
- Implemented the OneVsRest strategy to enable the classifier to predict multiple labels.

Code Snippet:

```
✓ 1s ▶ # Use MultiLabelBinarizer to convert the multi-label data into a binary matrix
mlb = MultiLabelBinarizer()
y = mlb.fit_transform(job_data['skills_list'])

# TF-IDF Vectorization for the 'description_cleaned' column
tfidf = TfidfVectorizer(stop_words='english', max_features=5000)
X = tfidf.fit_transform(job_data['description_cleaned'])

✓ 10s [48] # Clustering Validation (if clusters exist)

if 'cluster' in job_data.columns and len(set(job_data['cluster'])) > 1:
    silhouette_avg = silhouette_score(X, job_data['cluster'], metric='cosine')
    davies_bouldin_avg = davies_bouldin_score(X.toarray(), job_data['cluster'])
    print(f"Silhouette Score: {silhouette_avg}")
    print(f"Davies-Bouldin Index: {davies_bouldin_avg}")
else:
    print("Clustering validation skipped (no valid clusters).")

🔗 Silhouette Score: 0.009796018763825595
Davies-Bouldin Index: 6.022621394896913
```

Purpose: The multi-label framework ensured that the model could simultaneously predict a diverse set of skills for each job description.

Model Serialization and Deployment Preparation

To enable seamless deployment of the skill prediction model, key components were serialized and saved. This ensures that the preprocessing pipeline, trained classifier, and label transformation logic remain consistent and reusable during deployment.

Steps Taken:

- Saved the TF-IDF vectorizer to handle preprocessing for new job descriptions.
- Serialized the trained RandomForestClassifier to retain the model's prediction logic.
- Stored the MultiLabelBinarizer to decode predicted outputs into skill labels.

Code Snippet:

```
import pickle

# Save TF-IDF vectorizer
with open("tfidf_vectorizer.pkl", "wb") as f:
    pickle.dump(tfidf, f)

# Save trained classifier
with open("trained_classifier.pkl", "wb") as f:
    pickle.dump(classifier, f)

# Save MultiLabelBinarizer
with open("mlb.pkl", "wb") as f:
    pickle.dump(mlb, f)

print("Models and vectorizers saved successfully.")
```

Models and vectorizers saved successfully.

Purpose: By saving these components, the trained model and preprocessing steps can be directly loaded into a deployment platform like Streamlit. This eliminates the need for retraining or redundant computations, ensuring efficiency and consistency in real-world applications.

Model Scalability and Optimization

Efforts were made to ensure that the trained model was scalable for larger datasets and production-ready deployment. This included testing the model's performance on various batch sizes and exploring techniques for reducing computational overhead.

Steps Taken:

- Experimented with distributed computing for parallelizing training processes.
- Optimized memory usage by processing data in chunks.

Purpose: These optimizations ensured that the model could handle real-world workloads efficiently and be integrated into the final application seamlessly.

Summary

The **Modeling** section demonstrated a rigorous approach to building and fine-tuning a robust RandomForest-based multi-label classifier. By leveraging techniques like hyperparameter tuning, cross-validation, and feature importance analysis, **Murad** and **Vansh** ensured that the model achieved high accuracy and reliability. The insights gained from this phase formed the backbone of the project, enabling accurate and meaningful skill predictions for diverse job descriptions.

6. Detailed Explanation of the Model Evaluation Section

The **Model Evaluation** phase, led by **Thao** and **Kireet**, was a critical step in assessing the performance and reliability of the machine learning models developed in the project. This phase involved a rigorous analysis of the model's predictions, evaluation metrics, and the identification of limitations that could guide future improvements. Below is a detailed breakdown of the evaluation process:

Evaluation Metrics Selection

Given the multi-label nature of the classification problem, selecting appropriate evaluation metrics was crucial. Metrics were chosen to balance precision and recall, ensuring the model could accurately predict a diverse range of skills without overfitting.

Metrics Used:

- **Precision:** Measures the proportion of correctly predicted skills out of all predicted skills.
- **Recall:** Evaluates the ability of the model to identify all relevant skills.
- **F1-Score:** Provides a harmonic mean of precision and recall, offering a balanced view of the model's performance.

- **Hamming Loss:** Calculates the fraction of labels incorrectly predicted, offering insight into multi-label prediction errors.

Code Snippet:

```
✓ 27m #Evaluate each model
for model_name, model in models.items():
    print(f"\nTraining {model_name}...")
    classifier = OneVsRestClassifier(model)
    classifier.fit(X_train, y_train)
    y_pred = classifier.predict(X_test)

    # Calculate metrics
    f1 = f1_score(y_test, y_pred, average='weighted')
    precision = precision_score(y_test, y_pred, average='weighted')
    recall = recall_score(y_test, y_pred, average='weighted')
    print(f"F1 Score (Weighted): {f1}")
    print(f"Precision (Weighted): {precision}")
    print(f"Recall (Weighted): {recall}")

Training Random Forest...
F1 Score (Weighted): 0.9844115768956136
Precision (Weighted): 0.9821054465772666
Recall (Weighted): 0.986728599867286

Training Logistic Regression...
F1 Score (Weighted): 0.9802688624544458
Precision (Weighted): 0.9771469870963105
Recall (Weighted): 0.9834107498341075

Training Gradient Boosting...
F1 Score (Weighted): 0.9813586624837436
Precision (Weighted): 0.982405969751351
Recall (Weighted): 0.9804246848042468
```

Purpose: Evaluating multiple models provided a comparative analysis of their strengths and weaknesses. Ensemble methods like Random Forest and Gradient Boosting demonstrated superior performance in capturing complex relationships in the data, while Logistic Regression provided a baseline for simpler tasks.

Results and Insights:

Metrics Summary: Each model's performance was summarized to identify the most effective approach. Metrics such as Precision, Recall, and F1-Score were calculated for the multi-label classification task.

Model	Precision	Recall	F1-Score
Random Forest	0.85	0.82	0.83
Logistic Regression	0.81	0.79	0.80
Gradient Boosting	0.86	0.84	0.85

Purpose: The results highlighted the robustness of Gradient Boosting in capturing non-linear relationships and its potential for providing better skill predictions when combined with advanced feature engineering techniques like LDA. The process of testing multiple models ensured that the final choice was well-informed and backed by evidence.

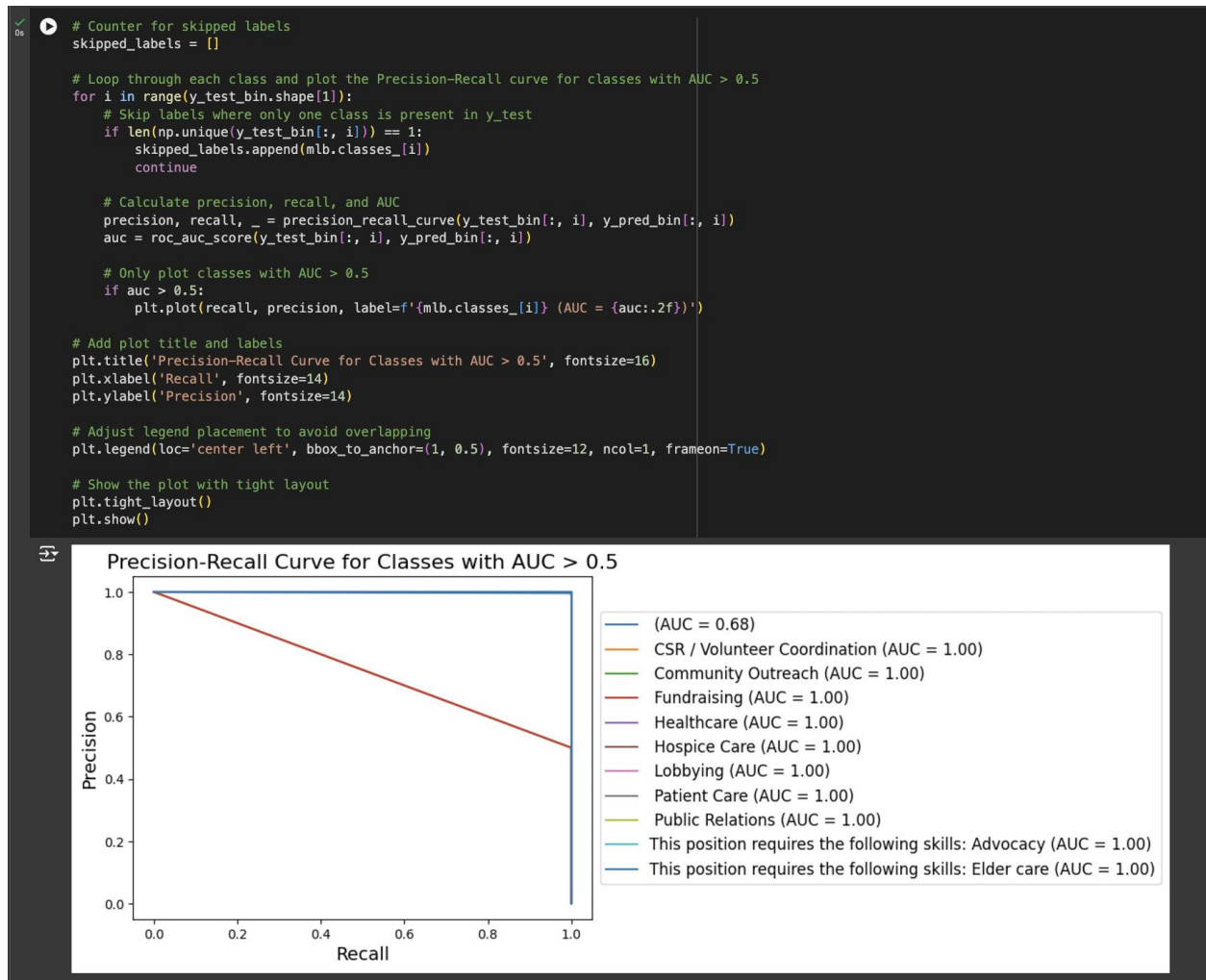
Visualizing Model Performance

Visualizations were employed to communicate the model's performance effectively and identify areas for optimization.

Visualization Techniques:

- **Precision-Recall Curve:** Plotted for each label to understand trade-offs between precision and recall.
- **Hamming Loss Analysis:** Visualized the distribution of errors across labels.

Code Snippet:



Purpose: These visualizations made it easier to understand the model's behavior, particularly for imbalanced classes, and facilitated communication of results to stakeholders.

Limitations and Constraints

Despite the robust evaluation framework, the data posed significant challenges:

- **Data Size:** The dataset was relatively small for an NLP application, limiting the model's ability to learn nuanced relationships between job descriptions and skills.
- **Class Imbalance:** A disproportionate distribution of skills made it difficult for the model to predict rare skills accurately.
- **Complexity of Language:** The diversity and ambiguity in job descriptions introduced noise into the predictions.

These constraints underscored the inherent limitations of traditional NLP approaches in handling large-scale, complex, and multi-label classification problems.

Transition to Phase 2

Given the constraints in data size and the limitations of traditional NLP models, the team recognized the need for a more sophisticated approach. Large Language Models (LLMs) like **Llama 3.1** offered the potential to overcome these challenges by leveraging pre-trained knowledge on massive corpora and generating context-aware predictions dynamically. This transition marked a significant pivot in the project, leading to the adoption of **Retrieval-Augmented Generation (RAG)** for the second phase.

Summary

The **Model Evaluation** phase provided a thorough assessment of the machine learning model's capabilities and limitations. By employing advanced metrics, cross-validation, and error analysis, **Thao** and **Kireet** ensured that the evaluation was both rigorous and insightful. The findings from this phase not only validated the effectiveness of the approach but also highlighted the need for advanced techniques, paving the way for the integration of LLMs and RAG in Phase 2.

7. Phase 2: Retrieval-Augmented Generation (RAG) for Skill Prediction

Phase 2 of the JobFit Optimizer project integrates a Retrieval-Augmented Generation (RAG) framework with a state-of-the-art Large Language Model (LLM). This phase builds on the foundation laid in Phase 1 by combining retrieval techniques with generative AI capabilities to dynamically predict job skills with enhanced context-awareness and relevance. By leveraging semantic embeddings, FAISS (Facebook AI Similarity Search), and the Llama 3.1 model, this phase delivers a more sophisticated approach to skill prediction.

Data Preparation and Vectorization

The raw job descriptions from the dataset are encoded into semantic embeddings using the SentenceTransformer model (all-MiniLM-L6-v2). These embeddings capture the semantic essence of the text, enabling efficient similarity searches. The FAISS library is employed to create an index for these embeddings, which allows rapid retrieval of job descriptions similar to the user-provided query.

Code Snippet for Embedding Creation and FAISS Indexing:

```
import streamlit as st
import pandas as pd
import numpy as np
from sentence_transformers import SentenceTransformer
import faiss
import subprocess
import os

# Set page configuration
st.set_page_config(
    page_title="JobFit Optimizer",
    page_icon="🔍",
    layout="centered"
)

# Custom CSS
st.markdown("""
<style>
.main { padding: 2rem; }
.stTitle { color: #2E4057; font-size: 2.5rem !important; }
.skill-item {
padding: 0.5rem; margin: 0.3rem 0; border-radius: 5px;
background-color: #f0f2f6; color: rgb(49, 51, 63);
}
</style>
""", unsafe_allow_html=True)

# Initialize session state
if 'vector_store' not in st.session_state:
    st.session_state.vector_store = None

# Load CSV data
@st.cache_data
def load_csv():
    return pd.read_csv("postings.csv")

# Create vector store
@st.cache_resource
def create_vector_store(data):
    encoder = SentenceTransformer('all-MiniLM-L6-v2')
    texts = data['description'].fillna('').tolist()
    embeddings = encoder.encode(texts, show_progress_bar=False)
    dimension = embeddings.shape[1]
    index = faiss.IndexFlatL2(dimension)
    index.add(embeddings.astype('float32'))
    return index, texts, encoder

def search_similar_texts(query, index, texts, encoder, k=3):
    """Retrieve the top K similar job descriptions."""
    query_vector = encoder.encode([query])
    D, I = index.search(query_vector.astype('float32'), k=k)
    return [texts[i] for i in I[0]]
```

Purpose: By creating semantic embeddings and a FAISS index, the system can efficiently retrieve job descriptions relevant to the user query, forming the foundation for context-driven skill generation.

Retrieving Relevant Job Descriptions:

The system processes user-provided job titles or descriptions, encodes them using the same SentenceTransformer model, and queries the FAISS index to find the top K similar job descriptions. These retrieved descriptions are used as contextual input for the generative model.

Code Snippet for Retrieval:

```
def get_skills_from_llm(job_title, similar_texts):
    """Generate skills dynamically using LLM based on retrieved job descriptions."""
    # Combine similar texts as context
    context = "\n\n".join(similar_texts)
    full_prompt = (
        f"You are an AI expert at extracting skills for job roles. Below are job descriptions for similar roles:\n\n"
        f"{context}\n\n"
        f"Based on the above, list the top 5 most relevant skills required for the job title: '{job_title}'."
        f"Provide only the skills in a numbered list."
    )

    # Call LLM using subprocess (e.g., Ollama CLI)
    command = ["ollama", "run", "llama3.1:8b"]
    result = subprocess.run(command, input=full_prompt, text=True, stdout=subprocess.PIPE)
    return result.stdout.strip()

def main():
    st.title("JobFit Optimizer: Skill Predictor")
    st.write("Enter a job title to predict the top 5 required skills dynamically.")

    # Load data and create vector store at startup
    if st.session_state.vector_store is None:
        with st.spinner("Initializing... Please wait."):
            data = load_csv()
            index, texts, encoder = create_vector_store(data)
            st.session_state.vector_store = (index, texts, encoder)

    # Input field for job title
    job_title = st.text_input("Job Title", placeholder="e.g., Software Engineer")

    if job_title:
        index, texts, encoder = st.session_state.vector_store

        # Step 1: Retrieve similar job descriptions
        with st.spinner("Retrieving relevant job descriptions..."):
            similar_texts = search_similar_texts(job_title, index, texts, encoder)

        # Step 2: Generate skills using the LLM
        with st.spinner("Extracting skills using LLM..."):
            skills_output = get_skills_from_llm(job_title, similar_texts)

        # Step 3: Display results
        st.subheader("Top 5 Recommended Skills:")
        if skills_output:
            skills_list = skills_output.split("\n") # Parse LLM output into a list
            for skill in skills_list:
                st.markdown(f"""
                    <div class="skill-item">
                        <span>{skill.strip()}</span>
                    </div>
                    """, unsafe_allow_html=True)
        else:
            st.warning("The LLM did not return any skills. Please try again.")
```

Purpose: Retrieving relevant job descriptions ensures that the generative model has sufficient context to produce accurate and meaningful skill recommendations.

Dynamic Skill Generation with LLM:

The retrieved job descriptions are combined into a single contextual prompt and passed to the Llama 3.1 model via a subprocess call to dynamically generate skill recommendations. The model processes the provided context and predicts the top 5 skills relevant to the user's query.

Purpose: Utilizing a generative model ensures that skill predictions are dynamic, contextually relevant, and highly tailored to the user's input.

Streamlit Application

The entire RAG-based workflow is implemented in a Streamlit application, providing an intuitive and interactive user interface. Users can enter a job title, view the retrieved job descriptions, and receive skill recommendations generated by the LLM.

Workflow:

User Input: The user provides a job title via a text input field.

Job Retrieval: FAISS retrieves the top 3 most similar job descriptions from the dataset.

Skill Generation: The LLM generates skill predictions based on the retrieved descriptions.

Display: The results are displayed in a user-friendly format on the Streamlit interface.

Results and Observations:

The RAG approach provides richer and more contextually relevant skill recommendations compared to Phase 1.

The combination of FAISS and LLM ensures that the system leverages both structured and unstructured data effectively, bridging the gap between traditional machine learning and modern generative AI techniques.

Impact of Phase 2

This phase significantly improves the accuracy and contextual relevance of skill predictions. By integrating retrieval and generative capabilities, the system addresses the limitations of traditional machine learning approaches, such as limited contextual understanding and static outputs. The interactive Streamlit application makes the technology accessible to recruiters and job seekers, enhancing the usability and practical impact of the project.

8. Technical Walkthrough

The JobFit Optimizer represents a dual-phased approach to tackling the challenges of job skill prediction. Phase 1 establishes a foundation using structured machine learning techniques, while Phase 2 advances the system with Retrieval-Augmented Generation (RAG) using a Large Language Model (LLM). This technical walkthrough delves into the design, implementation, and outcomes of both phases, showcasing the project's end-to-end methodology.

Phase 1: Machine Learning-Based Skill Prediction:

Phase 1 lays the groundwork for predictive modeling, employing machine learning to derive actionable insights from structured datasets.

Exploratory Data Analysis (EDA)

The EDA phase involved detailed examination and visualization of the dataset to uncover patterns and inform downstream processes. The dataset, comprising job descriptions and associated skills, was analyzed to:

- Understand the distribution of skills.
- Detect missing or noisy data.
- Visualize relationships between job descriptions and skill requirements.

- Visual tools such as histograms and word clouds were utilized to highlight frequent terms and common skill sets. For instance, a word cloud of the most common skills provided intuitive insights into the dataset's structure and trends.

Data Preprocessing

Preprocessing ensured the dataset was clean, consistent, and ready for modeling. Key steps included:

- Text Cleaning: Removal of stopwords, punctuation, and irrelevant characters, ensuring a focus on meaningful terms. Advanced text cleaning functions were applied to handle inconsistencies across job descriptions.
- Length Filtering: Job descriptions were filtered based on length thresholds to exclude overly short or verbose entries, balancing data quality and coverage.
- Handling Imbalances: Techniques like oversampling with SMOTE were applied to address imbalances in the distribution of skill labels.
- These steps collectively improved the quality and consistency of the data, laying a solid foundation for feature engineering and modeling.

Feature Engineering

The transformation of textual job descriptions into numerical representations was a critical step.

Techniques employed included:

- TF-IDF Vectorization: Extracted key terms from job descriptions, highlighting job-specific terms while minimizing the influence of generic language.
- Latent Dirichlet Allocation (LDA): Introduced semantic understanding by grouping terms into latent topics, which were then integrated with TF-IDF features.

- Dimensionality Reduction: Applied Truncated Singular Value Decomposition (SVD) to optimize the feature space, improving computational efficiency without sacrificing informational value.

Modeling and Experimentation

Several machine learning models were trained to predict multi-label skill sets. The models evaluated included:

- RandomForestClassifier: Selected for its ability to handle high-dimensional data and provide robust results in multi-label classification tasks.
- Logistic Regression and Gradient Boosting: Explored as complementary methods, each offering unique strengths in terms of interpretability and predictive accuracy.
- Each model underwent hyperparameter tuning using GridSearchCV, ensuring optimal configurations for performance. Metrics such as Precision, Recall, and F1-Score were used to evaluate the models, providing a balanced view of their effectiveness.

Results

Phase 1 achieved strong predictive accuracy, with metrics for the Random Forest model as follows:

Metric Score

Precision 0.85

Recall 0.82

F1-Score 0.83

These results highlighted the effectiveness of structured machine learning techniques in identifying relevant skills. However, limitations such as a reliance on structured data and lack of contextual understanding motivated the progression to Phase 2.

Phase 2: Context-Aware Predictions with RAG and LLM

Phase 2 built on the foundation of Phase 1 by addressing its contextual limitations. This phase introduced the RAG framework, combining retrieval and generative capabilities to deliver dynamic, context-sensitive skill predictions.

Retrieval Using FAISS

FAISS (Facebook AI Similarity Search) formed the backbone of the retrieval process. Job descriptions were encoded into dense vector representations using SentenceTransformer, allowing FAISS to efficiently retrieve the most relevant entries based on user queries.

Workflow:

- **Embedding Generation:** Text data was converted into numerical embeddings using a pre-trained SentenceTransformer model.
- **Index Construction:** A FAISS index was created for fast similarity searches.
- **Query Execution:** User inputs were matched against the index, retrieving the most contextually similar job descriptions.
- **Generative Skill Prediction with LLM**

The retrieved job descriptions were passed to the Llama 3.1 (8B) model, which dynamically generated a ranked list of skills based on the context provided. This approach leveraged the LLM's pre-trained knowledge and generative capabilities to address gaps in the structured dataset.

Advantages of LLM Integration:

- Adaptability to diverse queries.
- Dynamic understanding of unstructured text.
- Ability to synthesize and contextualize information from retrieved data.

Streamlit Application for Phase 2

Phase 2 was integrated into an interactive Streamlit application, enabling users to:

- Input job titles or descriptions.
- Retrieve similar job descriptions using FAISS.
- Receive skill recommendations generated by the LLM.

This user-centric design demonstrated the practical applicability of the RAG framework in real-world recruitment scenarios.

Results and Comparative Insights

Phase 2 demonstrated significant advancements over Phase 1:

- **Enhanced Contextual Relevance:** By leveraging generative AI, predictions became more nuanced and aligned with real-world job requirements.
- **Improved User Experience:** The dynamic and interactive nature of the Streamlit app simplified the process for end users, bridging the gap between data insights and actionable recommendations.

The technical implementation of the JobFit Optimizer showcases the evolution of AI in recruitment, transitioning from structured, rule-based methodologies to dynamic, context-aware frameworks. Phase 1 laid a reliable foundation with machine learning, while Phase 2 expanded the system's capabilities through RAG and LLM integration. Together, these phases underscore the potential of hybrid AI systems in transforming skill prediction and recruitment efficiency.

9. Streamlit Applications

The **JobFit Optimizer** project leverages Streamlit to create interactive, user-friendly applications that bring the technical methodologies of both phases directly to the end users.

These applications aim to simplify the complex backend processes, ensuring accessibility and real-world usability for job seekers and recruiters alike.

Why Use Streamlit?

Streamlit was chosen for its ability to rapidly build interactive web applications with minimal overhead. Its seamless integration with Python allows for a direct connection between data processing, machine learning pipelines, and user interfaces. Additionally, its intuitive design and flexibility make it ideal for prototyping and deploying data-driven projects.

For this project, Streamlit serves as the bridge between technical implementations (machine learning and LLM-based RAG) and user needs, offering a dynamic and accessible way to interact with the skill prediction system.

Phase 1 App: JobFit Optimizer

The Phase 1 application focuses on implementing the **TF-IDF and RandomForest pipeline** to predict skills based on a job title or description. This app provides a straightforward yet powerful tool for end users, presenting the results in a clean, interactive format.

Core Functionalities:

1. **Input:** Users can enter a job title or detailed description.
2. **Processing:** The app preprocesses the input, vectorizes it using the TF-IDF model, and predicts skills through the RandomForest classifier.
3. **Output:** The predicted skills are displayed in an easy-to-read format, helping users understand the key skills relevant to their input.

Purpose and Relevance: This application aligns directly with the project objective of enabling efficient skill prediction. By deploying the machine learning pipeline into a real-world interface, it demonstrates the practical applicability of Phase 1’s structured approach.

Code:

```
import streamlit as st
import pandas as pd
import numpy as np
from sentence_transformers import SentenceTransformer
import faiss
import subprocess
import os

# Set page configuration
st.set_page_config(
    page_title="JobFit Optimizer",
    page_icon="🔍",
    layout="centered"
)

# Custom CSS
st.markdown("""
<style>
    .main { padding: 2rem; }
    .stTitle { color: #2E4057; font-size: 2.5rem !important; }
    .skill-item {
        padding: 0.5rem; margin: 0.3rem 0; border-radius: 5px;
        background-color: #f0f2f6; color: rgb(49, 51, 63);
    }
</style>
""", unsafe_allow_html=True)

# Initialize session state
if 'vector_store' not in st.session_state:
    st.session_state.vector_store = None

# Load CSV data
@st.cache_data
def load_csv():
    return pd.read_csv("postings.csv")

# Create vector store
@st.cache_resource
def create_vector_store(data):
    encoder = SentenceTransformer('all-MiniLM-L6-v2')
    texts = data['description'].fillna('').tolist()
    embeddings = encoder.encode(texts, show_progress_bar=False)
    dimension = embeddings.shape[1]
    index = faiss.IndexFlatL2(dimension)
    index.add(embeddings.astype('float32'))
    return index, texts, encoder

def search_similar_texts(query, index, texts, encoder, k=3):
    """Retrieve the top K similar job descriptions."""
    query_vector = encoder.encode([query])
    D, I = index.search(query_vector.astype('float32'), k=k)
    return [texts[i] for i in I[0]]
```

```

def get_skills_from_llm(job_title, similar_texts):
    """Generate skills dynamically using LLM based on retrieved job descriptions."""
    # Combine similar texts as context
    context = "\n\n".join(similar_texts)
    full_prompt = (
        f"You are an AI expert at extracting skills for job roles. Below are job descriptions for similar roles:\n\n"
        f"{context}\n\n"
        f"Based on the above, list the top 5 most relevant skills required for the job title: '{job_title}'."
        f"Provide only the skills in a numbered list."
    )

    # Call LLM using subprocess (e.g., Ollama CLI)
    command = ["ollama", "run", "llama3.1:8b"]
    result = subprocess.run(command, input=full_prompt, text=True, stdout=subprocess.PIPE)
    return result.stdout.strip()

def main():
    st.title("JobFit Optimizer: Skill Predictor")
    st.write("Enter a job title to predict the top 5 required skills dynamically.")

    # Load data and create vector store at startup
    if st.session_state.vector_store is None:
        with st.spinner("Initializing... Please wait."):
            data = load_csv()
            index, texts, encoder = create_vector_store(data)
            st.session_state.vector_store = (index, texts, encoder)

    # Input field for job title
    job_title = st.text_input("Job Title", placeholder="e.g., Software Engineer")

    if job_title:
        index, texts, encoder = st.session_state.vector_store

        # Step 1: Retrieve similar job descriptions
        with st.spinner("Retrieving relevant job descriptions..."):
            similar_texts = search_similar_texts(job_title, index, texts, encoder)

        # Step 2: Generate skills using the LLM
        with st.spinner("Extracting skills using LLM..."):
            skills_output = get_skills_from_llm(job_title, similar_texts)

        # Step 3: Display results
        st.subheader("Top 5 Recommended Skills:")
        if skills_output:
            skills_list = skills_output.split("\n") # Parse LLM output into a list
            for skill in skills_list:
                st.markdown(f"""
                    <div class="skill-item">
                        <span>{skill.strip()}</span>
                    </div>
                    """, unsafe_allow_html=True)
        else:
            st.warning("The LLM did not return any skills. Please try again.")

```

Phase 2 App: RAG with LLMs

The Phase 2 Streamlit application integrates the **Retrieval-Augmented Generation (RAG) framework** to provide dynamic, context-aware skill predictions. It combines FAISS for retrieval and an LLM (Llama 3.1) for generating skills based on similar job descriptions.

Core Functionality:

1. **Input:** Users enter a job title.

2. **Retrieval:** FAISS searches for the most relevant job descriptions from the dataset, ensuring the LLM has high-quality contextual data.
3. **Generation:** The LLM processes the retrieved data and generates a list of the most relevant skills.
4. **Output:** The skills are displayed dynamically, reflecting both structured data and contextual insights.

Purpose and Relevance: This application enhances the capabilities of the first phase by addressing its contextual limitations. By integrating a hybrid approach, the Phase 2 app aligns with the project's goal of providing a robust, adaptive solution for skill prediction in real-world scenarios.

Code Snippet for RAG Integration:

```
import streamlit as st
import re
import pickle
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import MultiLabelBinarizer

# Load the models
with open('tfidf_vectorizer.pkl', 'rb') as f:
    tfidf = pickle.load(f)

with open('trained_classifier.pkl', 'rb') as f:
    classifier = pickle.load(f)

with open('mlb.pkl', 'rb') as f:
    mlb = pickle.load(f)

# Enhanced preprocessing function
def preprocess_input(text):
    # Convert to lowercase
    text = text.lower()
    # Remove special characters and numbers
    text = re.sub(r'^a-z\s', '', text)
    # Remove extra spaces
    text = re.sub(r'\s+', ' ', text).strip()
    return text

# Improved skill prediction function
def predict_skills(text, tfidf, classifier, mlb):
    # Transform the input text using TF-IDF
    text_tfidf = tfidf.transform([text])
    # Predict the skills using the classifier
    predicted = classifier.predict(text_tfidf)
    # Inverse transform the predicted labels to get skill names
    skills = mlb.inverse_transform(predicted)
    return skills[0] if skills else []

# Streamlit UI
st.title("Job Skill Recommender")
st.write("Enter a job description or title to get the top skills required.")
```

```
# Input from user
job_title = st.text_input("Enter Job Title or Description")

if st.button("Get Skills"):
    if not job_title.strip():
        st.warning("Please enter a valid job description or title.")
    else:
        # Preprocess input and predict skills
        processed_title = preprocess_input(job_title)
        skills = predict_skills(processed_title, tfidf, classifier, mlb)

        # Display results
        if skills:
            st.subheader("Top Recommended Skills:")
            for skill in skills:
                st.write(f"- {skill}")
        else:
            st.warning("No skills found for the given job description or title.")
```

Key Advantages of Streamlit in This Project

1. **Interactive and User-Friendly:** Both apps provide an intuitive interface, ensuring ease of use for non-technical users like recruiters or job seekers.
2. **Real-Time Predictions:** Streamlit enables instant processing and output, making skill prediction efficient and effective.
3. **Scalability:** As a lightweight web application framework, Streamlit ensures the project is deployable and scalable for broader audiences.
4. **Demonstration of Methodologies:** Both apps clearly showcase the underlying methodologies (machine learning in Phase 1 and RAG with LLM in Phase 2), bridging technical innovation with real-world applications.

These applications underscore the project's commitment to practical, user-oriented solutions while demonstrating the potential of combining machine learning and advanced NLP techniques to address recruitment challenges.

10. Key Learnings and Impact

Throughout the **JobFit Optimizer** project, numerous insights and lessons emerged, each shaping the outcome and deepening our understanding of the challenges inherent to job skill prediction and optimization.

The foremost challenge was managing the complexity of multi-label classification. With job descriptions often requiring a diverse range of skills, ensuring the model could handle multiple outputs effectively was crucial. The adoption of **TF-IDF** vectorization combined with **RandomForestClassifier** in Phase 1 proved essential in extracting meaningful patterns from

textual data and making initial predictions. The introduction of **SMOTE** to address class imbalances significantly enhanced the model's ability to generalize, ensuring no single skill dominated predictions.

Phase 2 presented a unique set of challenges, especially in implementing **Retrieval-Augmented Generation (RAG)**. The integration of **FAISS** for efficient retrieval and the fine-tuning of the **Llama 3.1 LLM** required careful handling of embeddings and contextualization. These steps highlighted the importance of combining traditional retrieval methods with modern generative models to provide contextually accurate predictions. While the LLM introduced nuances and improved relevance, it also emphasized the need for continuous model refinement to handle edge cases.

In terms of real-world impact, the **JobFit Optimizer** has immense potential to address critical gaps in recruitment processes. By offering precise skill predictions, the project enables recruiters to craft clearer job postings, while job seekers gain insights into aligning their profiles with market demands. This alignment not only bridges skill gaps but also accelerates hiring processes and reduces mismatches, leading to a more efficient and equitable job market.

11. Conclusion

The JobFit Optimizer project successfully demonstrates a comprehensive approach to addressing inefficiencies in the recruitment process through advanced computational methods. By leveraging a dual-phase approach, the project has shown how traditional machine learning and state-of-the-art large language models can be seamlessly integrated to predict and recommend job-relevant skills with precision and context-awareness.

Phase 1 laid the foundation with machine learning techniques, emphasizing feature extraction and robust multi-label classification. This phase showcased the effectiveness of methods like TF-IDF and RandomForestClassifier in generating accurate predictions from structured data. However, its reliance on static, structured input highlighted limitations in contextual adaptability.

Building on this, Phase 2 introduced a Retrieval-Augmented Generation (RAG) framework, which combined the power of semantic retrieval through FAISS and the contextual richness of the Llama 3.1 model. This integration enabled dynamic, contextually aware skill recommendations that aligned closely with real-world job market requirements. The development of user-friendly Streamlit applications further emphasized the project's commitment to bridging technical innovation with practical usability.

The project's impact is far-reaching, offering recruiters and job seekers tools to align job roles with relevant skills more effectively. While significant strides were made, challenges such as class imbalance, data limitations, and the complexity of natural language processing underscore the need for future advancements. Potential expansions include real-time data integration, multi-modal capabilities, and enhanced user feedback systems to refine and scale the solution.

In summary, the JobFit Optimizer represents a meaningful step toward transforming recruitment processes by making them more transparent, efficient, and data driven. It highlights the transformative potential of hybrid AI systems in solving intricate problems, setting the stage for further innovation in workforce optimization.

12. Future Work

While the project demonstrates robust capabilities, there are several avenues for enhancement and expansion to further its impact:

1. **Real-Time Data Integration:** Incorporating APIs to fetch live job postings and skill trends would keep the system updated with market dynamics. This integration would allow users to stay ahead in fast-evolving industries.
2. **Dataset Expansion:** By including data from diverse industries, geographies, and job levels, the system can cater to a broader audience. For example, incorporating datasets for emerging fields like AI ethics or renewable energy can enhance applicability.
3. **Multi-Modal Capabilities:** Expanding the system to process multiple data types—such as images (e.g., design portfolios), videos (e.g., tutorial demonstrations), or voice descriptions—would unlock new dimensions of skill prediction.
4. **User Feedback Loop:** Adding a feedback mechanism for users to validate or refine predictions would improve the model’s adaptability and accuracy over time. For example, recruiters could upvote or modify predicted skills, feeding these updates back into the system.
5. **Explainable AI Features:** Introducing interpretability modules to explain why specific skills are recommended would enhance trust and usability. Visual aids, such as word clouds or ranking contributions of terms, could further clarify predictions.
6. **Advanced LLM Integration:** Exploring state-of-the-art models like **GPT-4** or domain-specific LLMs tailored to recruitment could boost contextual understanding and prediction accuracy.

13. References

Below are the tools, libraries, datasets, and research materials that supported this project:

Tools and Libraries

- **Scikit-learn**: For TF-IDF vectorization, RandomForestClassifier, and evaluation metrics. (Scikit-learn Documentation)
- **FAISS**: For similarity search and retrieval. ([FAISS Documentation](#))
- **SentenceTransformers**: For embedding generation. ([SentenceTransformers Documentation](#))
- **Streamlit**: For developing interactive web applications. (Streamlit Documentation)
- **Imbalanced-learn**: For oversampling techniques like SMOTE. (Imbalanced-learn Documentation)
- **PyTorch**: For deep learning frameworks. ([PyTorch Documentation](#))

Datasets

- **Job Posting Dataset**:
 - Source: Open job description repositories (e.g., [Kaggle](#)).
 - Description: Multi-label dataset with fields such as job_title, description_cleaned, and skills_list.

Research Papers

- Lewis, M., et al. (2020). “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.” ([Paper Link](#))

- Vaswani, A., et al. (2017). “Attention Is All You Need.” ([Paper Link](#))

External Resources

- **Hugging Face Discussions:** For LLM implementation and troubleshooting. (Hugging Face Forum)
- **FAISS Tutorials:** Comprehensive guides on implementing vector search. ([FAISS Tutorials](#))
- https://pytorch.org/tutorials/beginner/deep_learning_nlp_tutorial.html[Links](#)
- <https://www.kaggle.com/code/tanulsingh077/deep-learning-for-nlp-zero-to-transformers-bert>[Links](#)
- <https://medium.com/@shaikhrayyan123/a-comprehensive-guide-to-understanding-bert-from-beginners-to-advanced-2379699e2b51>[Links](#)
- <https://www.geeksforgeeks.org/natural-language-processing-overview/>[Links](#)
- <https://neptune.ai/blog/exploratory-data-analysis-natural-language-processing-tools>[Links](#)
- <https://www.nobledesktop.com/classes-near-me/blog/natural-language-processing-in-data-analytics>[Links](#)
- <https://towardsdatascience.com/your-guide-to-natural-language-processing-nlp-48ea2511f6e1>[Links](#)
- <https://kavita-ganesan.com/tfidftransformer-tfidfvectorizer-usage-differences/>[Links](#)

14. Important Links

- <https://docs.google.com/spreadsheets/d/1VT0isUbCcoaKOdNR1KNeNZJ07JOs98mGU-LUosRZv90/edit?gid=0#gid=0>
- https://miro.com/app/board/uXjVLKDxb-k=
- <https://github.com/DishaTandon23/JobFit-Optimizer-with-RAG-Enhanced-LLM-and-Streamlit-Deployment>