Machine Learning Project (CM551PC)

LABORATORY PROJECT

ON

Resume Analyzer

Submitted

In partial fulfillment of the requirements for the award of the degree of

Bachelor of Technology in

Computer Science Engineering

(Artificial Intelligence & Machine Learning)

by

CHINNAMSETTI ROHAN 23261A6609

GUGULAVATH NAVEEN 23261A6615

KHYATI CHINTHA 23261A6620

Under the guidance of

Mrs. Jasthy Sreedevi

ASSISTANT PROFESSOR

DEPARTMENT OF EMERGING TECHNOLOGIES

MAHATMA GANDHI INSTITUTE OF TECHNOLOGY

Department of Emerging Technologies

MAHATMA GANDHI INSTITUTE OF TECHNOLOGY

GANDIPET, HYDERABAD-500075, INDIA

2025- 2026

MAHATMA GANDHI INSTITUTE OF TECHNOLOGY
(AUTONOMOUS)
(Affiliated To Jawaharlal Nehru Technological University Hyderabad)

GANDIPET, HYDERABAD-500075, Telangana

CERTIFICATE

This is to certify that the project entitled "**Resume Analyzer**" is being submitted by CHINNAMSETTI ROHAN bearing Roll No. 23261A6609, GUGULAVATH NAVEEN bearing Roll No. 23261A6615 and KHYATI CHINTHA bearing Roll No. 23261A6620 in partial fulfilment of the requirement for the **Machine Learning Laboratory Project** in **EMERGING TECHNOLOGIES** is a record of bonafide work carried out by them. The results of the investigations enclosed in this report have been verified and found satisfactory.

Mrs. Jasthy Sreedevi
Assistant Professor
ET Department
MGIT

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

In a hiring world where a single job opening can drown beneath a flood of applications, countless students watch their aspirations fade before they even reach a recruiter's screen—silently filtered out by automated systems that decide their fate within seconds. Determined to bridge this hidden divide, this project develops an AI-assisted resume analysis and job-role matching system designed to lift early-career applicants above this digital noise, especially in academic environments where computational power is limited. Rather than depending on heavy deep-learning architectures that dominate recruitment analytics, the system harnesses efficient classical machine-learning techniques—including SVM, Logistic Regression, KNN, Decision Tree, Naive Bayes, and Random Forest—trained on TF-IDF vectorized text features and enhanced with PCA for dimensionality reduction. The workflow integrates resume preprocessing, keyword extraction, job–skill alignment, cosine-similarity scoring, and K-Means clustering to identify suitable roles from a synthetically generated job dataset. Using stratified train–test evaluation, Random Forest consistently rises as the most effective model across accuracy, precision, recall, F1-score, and sensitivity. The findings reveal that lightweight ML approaches can still deliver powerful, reliable, and interpretable results for resume screening, job-role recommendation, and interview preparation—further strengthened by a STAR-based interview evaluation module—making this system both academically impactful and a compelling foundation for future recruitment-support solutions. Keywords: TF-IDF Vectorization, PCA, K-Means Clustering, Cosine Similarity, Random Forest, SVM, Logistic Regression, KNN, Decision Tree, Naive Bayes.

Keywords: TF-IDF Vectorization, PCA (Principal Component Analysis), K-Means Clustering, Cosine Similarity, Random Forest, Support Vector Machine (SVM), Logistic Regression, K-Nearest Neighbors (KNN), Decision Tree, Naive Bayes

1. INTRODUCTION

In today's fast-moving job market, it often feels as though opportunities disappear long before students even realize they had a chance. A single job posting can attract hundreds—sometimes thousands—of applicants within hours, and before any human being reviews them, most resumes are silently screened out by automated filters **[1][2]**. For many fresh graduates, this creates a frustrating and discouraging cycle: they invest significant time preparing resumes, searching for roles, and practicing interview answers, yet receive almost no feedback to help them improve. Over time, this uncertainty becomes one of the biggest barriers for students entering the workforce—not because they lack skill or ambition, but because they lack the guidance needed to navigate these early steps. This project emerged from that gap: the need for a practical, student-friendly system that demystifies resume screening and gives aspiring candidates a clearer understanding of where they stand, what they need to improve, and which roles genuinely match their potential.

Recruitment practices have evolved rapidly over the past decade. Employers now manage an immense volume of applications and rely heavily on automated tools—keyword scanners, ranking algorithms, and quick filtering mechanisms—to shortlist candidates in seconds **[3][4]**. While these systems enable companies to handle scale, they often overlook subtle yet meaningful aspects of a student's background, such as academic projects, transferable abilities, or early technical experience. As a result, many students are rejected before a recruiter ever reads a single line of their profile. At the same time, first-time job seekers struggle to identify roles that align with their developing skills, and they rarely receive structured guidance on how to present themselves in interviews **[5]**. These challenges highlight a broader issue: existing tools might scan resumes, but they do little to explain, mentor, or support candidates through the full preparation process.

This project seeks to bring that support into one integrated platform. Rather than treating resume analysis, job-role prediction, and interview practice as isolated tasks, the system connects them into a unified workflow built specifically for students **[6]**. Users upload their resumes, receive clear insights about their skills and experience, discover which job roles best match their profile, and even practice interviews with structured feedback based on the STAR method **[7]**. The aim is to replicate the kind of personalized mentorship that students often need but rarely have access to, especially during the crucial early phases of their job search.

1.1 Problem Definition

One of the most persistent challenges students face when entering the recruitment process is not a lack of effort—it is a lack of clarity. Applicants refine their resumes, apply to multiple roles, and prepare for interviews, yet most hear nothing back. Automated filters used by companies operate primarily on strict keyword rules **[8]**, often ignoring the broader context of a student's capabilities. At the same time, many students are unsure which job titles actually fit their strengths or what skills they are missing. The core problem, therefore, is the absence of a single, accessible system that analyzes a student's resume, provides personalized job-role insights, and offers structured interview guidance in a way that is easy to understand and act upon **[9]**.

1.2 Objectives

This project is designed with the following goals to enhance the early-stage recruitment experience for students:

- To automatically read and analyze resumes using simple yet reliable text-processing methods **[1][3]**.
- To extract meaningful details—including skills, keywords, and estimated experience—directly from the resume **[2]**.
- To compare a candidate's profile with multiple job roles using ML-based similarity measures **[4][7]**.

- To train machine-learning models capable of predicting suitable job titles **[6][9]**.
- To highlight mismatches and missing skills, enabling students to identify areas for improvement **[5]**.
- To provide a STAR-based mock-interview module that evaluates student responses and delivers constructive feedback **[7]**.
- To create a unified platform that assists students from resume evaluation all the way to interview preparation [**10]**.

1.3 Existing Approaches

Most tools currently available to students work in isolation and offer only surface-level insights. Resume scanners tend to rely purely on keyword presence **[1][8]**, ignoring context and overlooking nuanced academic experience. Job portals often suggest roles that are too broad or irrelevant to a student's developing profile **[4]**. Interview-practice tools typically provide questions but do not evaluate the quality of answers in a structured manner **[5]**. Because these systems operate independently rather than as a cohesive unit, students receive fragmented guidance that fails to address their full preparation needs **[9][10]**. These limitations underscore the need for a more integrated, feedback-driven solution.

1.4 Proposed Approach

The system proposed in this project brings together resume analysis, job-role prediction, and interview preparation into one coherent framework designed with students in mind. The platform begins by extracting text from the uploaded resume and identifying relevant skills, keywords, and experience indicators **[1][2]**. These features are analyzed against a synthetic job dataset using TF-IDF vectorization **[2][3]**, PCA for dimensionality reduction **[3][4]**, cosine similarity **[1][5]**, and K-Means clustering **[2][9]** to generate match percentages and identify skill gaps. To improve prediction accuracy, machine-learning models such as Random Forest, SVM, Logistic Regression, KNN, Decision Tree, and Naive Bayes are trained and evaluated using standard metrics

[6][7][9]. The system also includes a mock-interview module that evaluates answers using the STAR method **[5][7]**, providing students with clear feedback on how effectively they communicate their experiences. While the current implementation uses synthetic datasets and classical ML techniques, it demonstrates how multiple stages of the recruitment process can be combined into a single, student-supportive tool **[10]**.

1.5 Requirements Specification

For a system that involves resume parsing, text processing, machine-learning inference, and interactive user interfaces, defining the correct software and device requirements is essential. Ensuring the right environment allows the platform to run efficiently, maintain reliability, and support future expansion.

1.5.1 Software Requirements

- Python – Core development language for text extraction, preprocessing, ML pipelines, and similarity calculations **[3][7]**.
- Streamlit or Flask – Lightweight frameworks for building a user-friendly interface where students upload resumes and view results **[10]**.
- Scikit-learn – Implements TF-IDF, PCA, clustering, and all classification algorithms including Random Forest, SVM, Logistic Regression, KNN, Decision Tree, and Naive Bayes **[6][9]**.
- Pandas and NumPy – Handle data manipulation, feature extraction, and dataset organization **[3]**.
- Matplotlib / Seaborn – Generate bar charts, match-score visuals, and skill-gap graphs **[5]**.
- PDFMiner, PyPDF2, or python-docx – Extract readable text from PDF and DOCX resumes **[1][4]**.
- Jupyter Notebook / VS Code / PyCharm – Support model experimentation, debugging, and complete development workflows.

1.5.2 Target Devices

- Desktop or Laptop with moderate CPU performance.
- Any device supporting Python or a browser-based deployment.
- Stable internet connection for online versions.
- Modern web browsers like Chrome, Edge, or Firefox.
- Optional GPU support for future upgrades involving advanced NLP or deep learning.

2. LITERATURE SURVEY ON RESUME ANALYZER

[1] AI-Based Resume Ranking System Using TF-IDF and Similarity Matching (2025)

When speed matters more than sophistication, a simple idea can sometimes outperform an entire toolkit. Afrin Shaikh's study follows this principle by using TF-IDF and cosine similarity to score how closely resumes match job descriptions. The approach focuses on transparency—each ranking is based on clear, traceable keyword patterns, making it easy for HR teams to understand why a candidate scored higher or lower. While the system works efficiently for straightforward text comparisons, it struggles when wording varies or context becomes important, since it relies solely on lexical overlap. Even so, the paper highlights how a clean, lightweight model can provide dependable first-round filtering without requiring heavy computation or advanced machine-learning pipelines.

[2] Resume Clustering and Job Description Matching Using TF-IDF + K-Means (2025)

In large hiring cycles where resumes pile up quickly, organizing them into meaningful groups becomes crucial. Ugle and colleagues address this by applying TF-IDF with K-Means clustering to group candidates according to shared skills and domain keywords. This method helps recruiters navigate high-volume applicant pools by revealing clusters that naturally emerge within the data, making it easier to identify who fits particular job categories. While the system offers a practical way to reduce manual screening effort, it remains sensitive to the limitations of TF-IDF—particularly when applicants phrase similar skills in different ways. Still, the research shows how unsupervised clustering can bring order and clarity to chaotic resume collections.

[3] Automated Resume Screening for HR Using Machine Learning (2025)

As hiring pipelines grow more demanding, supervised learning has become a powerful tool for automating candidate classification. This study from ARISET explores models like Logistic Regression, SVM, and Random Forest, all trained on labeled resume datasets to predict job roles with strong accuracy scores. The work pays special attention to feature importance, revealing which parts of a resume carry the most weight when the model makes decisions. Although the system performs well for clean, structured resumes, it struggles with unusual layouts and context-heavy descriptions due to its dependence on TF-IDF features. Nevertheless, the paper demonstrates how supervised ML can drastically streamline the early stages of HR screening.

[4] Resume Classification Using Machine Learning for Web & Mobile Systems (2025)

With more organizations integrating automated screening tools into mobile and web platforms, speed and stability are just as important as accuracy. This study focuses on building a lightweight classification system using TF-IDF along with Logistic Regression and Random Forest, optimized for real-time deployment. The authors highlight how their model maintains reliable performance across key metrics while remaining efficient enough to run on low-resource environments like mobile backends. Its main limitation lies in its reliance on lexical patterns, which weakens its performance when resumes contain creative formatting or non-standard phrasing. Even so, the paper offers a practical blueprint for scalable, production-ready screening systems.

[5] Intelligent Resume Parsing and Ranking System (2025)

In recruitment settings where resumes arrive in countless styles and formats, structured parsing becomes essential. This paper from IJRPR introduces a pipeline that extracts key fields from PDF and Word documents using rule-based patterns, then applies TF-IDF to rank candidates against job descriptions. The system excels at transforming disorganized text into structured, searchable information, giving recruiters clearer insights into candidate qualifications. However, its rule-driven parsing is fragile—unusual resume designs can cause extraction errors, and the method lacks deeper semantic understanding. Despite this, the study shows how an end-to-end architecture can turn raw documents into meaningful, ranked outputs suitable for downstream hiring decisions.

[6] Resume Screening with NLP in Turkish Using TF-IDF and Similarity Metrics (2025)

Screening resumes in non-English languages introduces unique linguistic challenges, and this study by Saatçı, Kaya, and Ünlü addresses them head-on. Working with Turkish resumes, the authors design a language-specific preprocessing pipeline including custom stopwords, morphological normalization, and localized tokenization. By combining these steps with cosine and Jaccard similarity, the system produces more accurate and fair matches than English-oriented tools could provide. Limited dataset size and the absence of modern semantic models keep the results from achieving deeper contextual understanding, but the study stands out for showing how tailored NLP can significantly improve screening quality in regional labor markets.

[7] Optimizing Job Matching Through Automated Screening (IEEE, 2024)

In an era where hiring decisions depend heavily on information retrieval, this IEEE paper examines how TF-IDF weighting and similarity metrics can refine job-candidate matching. The authors experiment with different textual features—skills,

education, experience—to determine which ones most strongly influence match quality. Their findings show that smart feature engineering can noticeably enhance precision and recall when compared with simple keyword searches. While the study stops short of exploring semantic embeddings or dimensionality reduction techniques, it successfully demonstrates how classical IR methods remain powerful when thoughtfully applied to real-world recruitment scenarios.

[8] Automatic Resume Screening Using SBERT Embeddings (2025)

As hiring tools evolve, semantic understanding has become essential, and this IJFMR study leverages SBERT to bridge that gap. By converting resumes and job descriptions into dense vector embeddings, the system identifies deeper relationships between concepts—even when different words are used to express similar skills. This allows for more natural and context-aware matching, outperforming TF-IDF in most test cases. The trade-off comes in the form of higher computational demands, especially at scale, and the need for GPU-friendly infrastructure. Despite these constraints, the paper clearly illustrates how embedding-based systems can reshape the future of automated recruitment.

[9] AI-Powered Resume Analyzer with Clustering and SVM Classification (2025)

When resume datasets grow large and diverse, combining unsupervised and supervised learning can provide a clearer picture of candidate profiles. This IJMRSET study uses TF-IDF with K-Means to cluster applicants by skills, followed by an SVM classifier to assign them to specific hiring categories. The hybrid approach helps organizations both explore natural groupings in the data and enforce consistent classification rules. While the clustering step struggles with imbalanced datasets and lacks dimensionality reduction, the system still demonstrates how mixing exploratory and predictive methods can deliver a more organized screening workflow.

[10] Comprehensive Analysis of Resume Recommendation Systems (2023)

Some studies aim to innovate; others aim to understand the entire landscape—and this work by Elsheh and colleagues does the latter. The authors compare a wide spectrum of resume-matching techniques, from TF-IDF and POS weighting to clustering and classical ML classifiers, evaluating how each contributes to overall recommendation accuracy. Their results show that hybrid pipelines consistently outperform single-method approaches, especially when feature normalization and skill-recency weighting are included. Although the paper doesn't dive into transformer-based embeddings, it provides a strong analytical foundation explaining why multi-layered systems tend to capture candidate suitability more effectively than simple frequency-based matching.

LITERATURE SURVEY SUMMARY

Year	Author(s)	Title	Dataset Used	Preprocessing Techniques	Metrics reported	Advantages	Limitations
2025	Afrin Shaikh	AI-Based Resume Ranking System Using TF-IDF and Similarity Matching	Resumes + Job Descriptions (from application)	TF-IDF, Cosine similarity	Relevance / ranking score (1–10)	Lightweight, interpretable, easy to deploy	Does not use clustering, PCA, or supervised classification (only similarity ranking) (IJSREM)
2025	Archana V. Ugale, Gayatri Sanap, Rutik Gunjal, Amit Ghumare, Shreyas Andhale	Resume Clustering and Job Description Matching	Resume corpus (IRJMETS, India) (IRJMETS)	TF-IDF, clustering (K-Means), matching (cosine, semantic)	Clustering metrics / matching score	Clusters similar resumes to reduce HR workload; uses clustering + matching pipeline	Does not report use of PCA; supervised classification missing; possible lack of standard dataset
2025	(IARJSET)	Automated Resume Screening for HR Using Machine Learning	Labeled resume dataset (IT roles) (IARJSET)	TF-IDF vectorization, ML: Random Forest, SVM, Regression	Accuracy (> 85%)	Real-world classification; reduces human bias; uses multiple classifiers	No clustering, no PCA; limited feature engineering details; specificity to IT roles
2025	(IJIRSET)	Resume Classification using Machine Learning	Resumes labeled by job-role (IJIRSET data) (IJIRSET)	TF-IDF, supervised learning: Logistic Regression, Random Forest	Accuracy, precision, recall, F1	Simple pipeline; interpretable models; scalable via web app	No clustering or dimensionality reduction; no unsupervised or ensemble beyond RF

2025	(IJRPR)	An Intelligent Resume Parsing and Ranking System	Resume documents (PDF, Word) (IJRPR)	NLP parsing, TF-IDF, similarity measures, supervised ranking (ML)	Matching / ranking accuracy, possibly precision/recall	End-to-end parsing + ranking system; practical integration	Doesn't explicitly mention PCA or clustering; limited ML algorithm variety described
2025	Mehtap Saatçı, Rukiye Kaya, Ramazan Ünlü	Resume Screening with Natural Language Processing (NLP)	123 job positions' resumes (Turkish) (DergiPark)	Preprocessing (tokenization, stopword), TF-IDF, similarity (Cosine, Jaccard)	Similarity scores, evaluation of matching	Focus on fairness, bias reduction, interpretable similarity matching	No clustering, no dimensionality reduction, no supervised classification (or limited)
2024	(IEEE)	Optimizing Job Matching Through Automated Resume Screening	Real / simulated resumes (conference paper) (DOI)	TF-IDF, possibly other ML (they mention screening via ML)	Accuracy, precision, recall, F1 (typical)	Focus on optimizing screening process; likely uses a mix of ML models	The paper abstract does not clearly mention clustering or PCA; details limited to screening task
2025	(IJFMR)	Automatic Resume Screening Using SBERT	Resumes + job descriptions (semantically) (IJFMR)	Sentence BERT embeddings, Cosine similarity, ML scoring, LLM-based recommendation	Suitability score, ranking metrics	Uses semantic representation (SBERT) for better matching than just TF-IDF; modern approach	Does not use TF-IDF, PCA, clustering; uses deep embeddings instead
2025	(IJMRSET)	AI-Powered Resume Analyzer with Clustering	Resume dataset from recruiters / application	TF-IDF, K-Means clustering, word embeddings,	Accuracy, precision, recall, silhouette score	Combines clustering + TF-IDF + embeddings; helps in	Does not clearly mention PCA; possible limitation in

			pool (IJMRSET)	feature extraction		resume grouping and job recommendation	handling resume format variety
2023	E. Elsheh et al.	Analysis of Resumes and Recommendations for Job Matches	Resume dataset labeled by job-role (Libyan journal) (ljast.ly)	TF-IDF, part-of-speech weighting, clustering, similarity (cosine, Jaccard, others)	Recommendation quality, similarity metrics	Good combination of TF-IDF + clustering + advanced similarity; richer matching than just frequency	No use of PCA; machine learning classifiers (RF/SVM etc.) not central; focus more on matching than classification

Table 2.1: Literature Survey Summary

3. DESIGN AND METHODOLOGY

It is surprising how many students send out dozens of applications without ever understanding why a few resumes move forward while most are rejected within seconds **[1][2]**. That silent uncertainty—unseen, unexplained, and frustrating—became the motivation behind this project. Instead of leaving applicants to guess what went wrong, the aim was to build a system that breaks the hiring pipeline into clear, understandable steps **[3]**. What began as a simple resume checker grew into a structured, AI-assisted tool that evaluates resumes, identifies suitable job roles, highlights skill gaps, and even helps users prepare for interviews **[4][5]**. The goal is simple: turn an opaque and overwhelming process into one that is transparent, supportive, and genuinely useful for early-career applicants **[6]**.

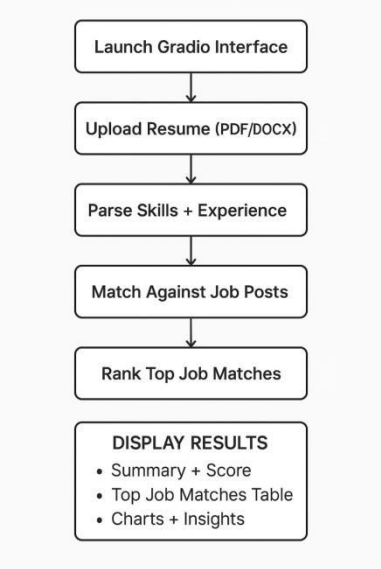

Figure 3.1: Resume Analysis & Job Matching Workflow

3.1 Flowchart Explanation

The workflow of the system follows a clear, step-by-step sequence **[1]**, and the flowchart represents how each module contributes to the overall analysis. Every stage serves a distinct purpose, yet all components work together to deliver a complete evaluation of the user's resume.

Step 1: Launch Gradio Interface

The process begins when the user opens the Gradio interface. At this point, the system loads all essential backend elements—including the TF-IDF vectorizer, PCA model, trained machine-learning classifiers, scoring functions, and the synthetic job-description dataset—so that the application is ready for immediate use **[2][3]**.

Step 2: Upload Resume

The user uploads a resume in PDF, DOCX, or TXT format. Depending on the file type, PyPDF2, python-docx, or standard text parsing is used to extract the content. The extraction module cleans the text to ensure it is consistent, readable, and suitable for further processing **[4]**.

Step 3: Parse Skills and Experience

The cleaned text is examined for technical skills, tools, soft skills, and experience indicators. This is done using a combination of predefined skill dictionaries, keyword rules, and regular expressions. The output is a structured profile of the candidate's abilities **[5]**.

Step 4: TF-IDF Vectorization

The resume text is transformed into numerical features through TF-IDF. Job descriptions in the dataset undergo the same vectorization process so that both representations exist on a comparable scale **[1][2]**.

Step 5: Apply PCA

Principal Component Analysis reduces the high-dimensional TF-IDF vectors into a more compact form. This step removes noise, improves efficiency, and preserves the most important informational patterns needed for matching **[3][7]**.

Step 6: Job Matching Using Cosine Similarity

The reduced resume vector is compared against the job-description vectors. Cosine similarity is used to measure how closely the candidate's profile aligns with a given role **[1][8]**:

$$\text{Similarity}(A,B) = (A \cdot B) / (||A|| \times ||B||)$$

Higher similarity values indicate stronger alignment between the resume and a job role.

Step 7: Rank Top Matches

Similarity results are combined with skill overlap and domain alignment to generate final match percentages. The system organizes the roles from strongest to weakest match, highlighting the most relevant opportunities **[6]**.

Step 8: Generate Visualizations

Visual summaries are created using Matplotlib or Plotly. These include bar charts, pie charts, tables, and score indicators, making the results easier for users to interpret **[5]**.

Step 9: Display Results

All outputs—job recommendations, similarity scores, resume strength, and identified skill gaps—are presented through the Gradio interface in a clear, user-friendly layout **[10]**.

Step 10: Mock Interview Module

To support interview preparation, the system presents technical or behavioral questions. User responses are evaluated through the STAR (Situation–Task–Action–Result) method, and the feedback highlights which parts of the answer are strong and where clarification or detail is needed **[9]**.

Overall, the flowchart captures this structured pipeline from resume upload to final interview feedback. Each stage is designed to be modular and efficient, ensuring the system is easy to maintain, extend, and refine.

3.2 System Overview

The system is organized into three core components that work together to deliver a complete, end-to-end recruitment-support workflow**[3][6]**:

1. Data Preparation and Job Dataset Generation: This module creates and structures the synthetic job dataset, defines domain-skill mappings, and prepares the information needed for matching.

2. Resume Processing and Job-Matching Engine: This component extracts text from the uploaded resume, identifies skills and experience, converts the content into numerical representations, and computes job-match scores using machine-learning and similarity-based methods.

3. Mock-Interview and STAR-Based Evaluation Module: The final module supports interview readiness by generating questions and evaluating user responses through the STAR (Situation–Task–Action–Result) framework **[9]**.

Although each module performs its tasks independently, they interact seamlessly through a unified Gradio interface. This integration ensures a smooth user experience while keeping the system modular, easy to debug, and flexible for future enhancements **[10]**.

3.3 Dataset Design and Preparation

One of the central challenges in building a job-recommendation system is obtaining a clean, standardized, and sufficiently large dataset of job descriptions **[1][6]**. Real-world job data is often inconsistent, proprietary, or incomplete, making it unsuitable for controlled experimentation. To address this limitation, the system relies on a carefully structured synthetic dataset that imitates real hiring trends while ensuring consistency across all evaluation stages **[2]**.

3.3.1 Job Category and Skill Mapping

At the core of the dataset is a domain–skill dictionary designed to reflect realistic industry expectations **[3]**. Each job category—such as Data Science, Backend Development, QA/Testing, Web Development, and DevOps—includes the tools, technologies, and competencies commonly associated with that field.

This mapping serves several key purposes:

- It allows the system to automatically generate synthetic job postings that resemble real listings **[2]**.
- It provides the criteria needed to compare a user's resume against industry-standard skill requirements **[3]**.
- It enables the system to detect missing or underrepresented skills in the user's profile **[5]**.
- It supports clustering of job descriptions into meaningful domains, improving recommendation accuracy **[7]**.

By defining this structured relationship between skills and job roles, the system ensures uniformity across all stages of processing.

3.3.2 Synthetic Job Posting Generation

A custom-built DataLoader module uses the domain–skill dictionary to generate thousands of job entries **[2][6]**. Each entry contains attributes typically found in real employment listings, including:

- Job Title
- Required Skills
- Company Name
- Salary Range
- Location
- Company Tier
- Job Type (On-site, Remote, Hybrid)
- Tools and Technologies

Although the dataset is artificial, it is constructed to mirror patterns observed in actual job markets—such as varying skill combinations, differences in company tiers, and realistic salary ranges. This ensures that the recommendation engine behaves reliably and produces results that reflect real-world hiring expectations.

3.4 Resume Parsing Module

The resume parsing module forms the foundation of the system's analytical process. Its purpose is to convert a user's resume—regardless of its original format—into a structured representation of skills, qualifications, and experience. This structured output is what later feeds into the job-matching and recommendation engine.

3.4.1 File Reading

The system is designed to accept multiple resume formats to ensure accessibility for all users **[4]**. It supports:

- PDF files, extracted using PyPDF2,
- DOCX documents, processed through python-docx, and
- TXT files, read directly through standard text parsing.

Regardless of the format, each resume is converted into clean, machine-readable text. Formatting artifacts such as bullet symbols, irregular spacing, headers, or non-UTF

characters are removed during this stage. This normalization ensures that downstream modules receive consistent, noise-free text for analysis.

3.4.2 Skill Extraction and Experience Estimation

Once the raw text has been extracted and cleaned, the system begins identifying key informational components of the resume **[5][6]**. This is done using a rule-based approach that combines:

- Keyword mapping (linked to the predefined domain–skill dictionary),
- Regular expressions (for detecting years of experience, dates, certifications, and project structures), and
- Simple pattern-recognition techniques for structured sections like education and projects.

From the text, the system extracts **[5]**:

- Technical skills
- Tools, libraries, and frameworks
- Soft skills
- Certifications
- Academic and personal projects
- Educational qualifications
- Estimated years of experience

This transparent, explainable extraction method ensures that the system's behavior remains academically verifiable and free from "black-box" decision-making.

3.5 Machine-Learning Model Training

The system's predictive capability relies on a structured machine-learning pipeline designed to classify job roles, rank matches, and quantify resume quality. To accomplish this, multiple classical ML algorithms were trained on the synthetic job

dataset, supported by vectorization, dimensionality reduction, clustering, similarity scoring, and a transparent weighted evaluation model. Each component contributes to making recommendations more accurate, interpretable, and computationally efficient **[1][2][3][6]**.

3.5.1 Algorithms Used

A diverse collection of lightweight machine-learning models was trained to classify job titles based on TF-IDF features **[3][6][7]**. The models include:

- Random Forest
- Support Vector Machine (SVM)
- Logistic Regression
- K-Nearest Neighbors (KNN)
- Decision Tree
- Naive Bayes

These algorithms were intentionally selected because they perform well on short text, handle sparse TF-IDF vectors effectively, and run efficiently even on limited hardware—making them suitable for an academic environment. Training multiple models also allows comparison across architectures to identify which classifier captures domain patterns most reliably.

3.5.2 TF-IDF Vectorization

To make resume text and job descriptions comparable, both are transformed into numerical vectors **[1][2]** using TF-IDF. This method highlights domain-specific terms and reduces the weight of common words.

TF(t,d) = Frequency_of_t / Total_terms_in_d

IDF(t) = log(N / n_t)

TFIDF(t,d) = TF(t,d) * IDF(t)

Where:

- N = total number of documents
- n_t = number of documents containing term

TF-IDF ensures that technical skills such as Python, Selenium, or SQL receive higher importance than generic terms.

3.5.3 Dimensionality Reduction (PCA)

Since TF-IDF generates high-dimensional and sparse vectors, Principal Component Analysis (PCA) is applied to compress these features while retaining essential information **[3][7]**.

Benefits of PCA:

- Faster similarity calculations
- Reduction of noise in sparse vectors
- Improved clustering stability
- Lower memory consumption

The resulting compact feature vectors improve the efficiency of both the classifier and similarity computation modules.

3.5.4 Clustering with K-Means

To organize job descriptions into meaningful groups, the system uses the K-Means clustering algorithm, producing domain-specific clusters such as **[2][7][9]**:

- Machine Learning / Data
- Software Development
- Testing / QA
- Cloud & DevOps
- Web Technologies

Cluster assignment is computed using: $\text{Cluster}(x) = \arg\min_k || x - \mu_k ||^2$

Where μ_k is the centroid of cluster .

Clustering ensures that each resume is compared primarily with relevant job categories, improving recommendation precision.

3.5.5 Similarity Computation Using Cosine Similarity

Cosine similarity serves as the core metric for determining how closely a resume aligns with each job description **[1][8]**:

$\text{Similarity}(A,B) = (A \cdot B) / (||A|| \times ||B||)$

Interpretation:

- 0.8–1.0 → Excellent match
- 0.5–0.7 → Moderate match
- 0.2–0.4 → Weak match

Roles with higher similarity scores are ranked at the top of the recommendation list.

3.5.6 Combined Resume Scoring Model

In addition to matching, the system assigns each resume a transparent score out of 100, based on weighted criteria that reflect typical ATS assessments **[6]**:

Component	Weight
Skill Match	40%
Experience	20%
Projects	15%
Education	10%
Certifications	5%
Tools/Technologies	10%

The final score is computed as: $\text{Score} = \Sigma (w_i \cdot s_i)$

Where w_i is the weight and the normalized score for each category.

This scoring system gives users a clear understanding of résumé strengths and improvement areas.

3.5.7 Model Performance Evaluation

Each classifier was evaluated using standard metrics to measure its ability to handle short-text job titles and domain-specific vocabulary **[3][6][7][9]**.

Accuracy = (TP + TN) / (TP + TN + FP + FN)
Precision = TP / (TP + FP)
Recall = TP / (TP + FN)
F1 = 2 * (Precision * Recall) / (Precision + Recall)

These metrics help identify how well each model distinguishes between job categories, especially within an imbalanced dataset.

Findings:

Across all evaluations, Random Forest **[6]** outperformed the other models in accuracy, precision, recall, F1-score, and sensitivity. Its ability to capture nonlinear relationships and handle noisy TF-IDF vectors made it the most reliable classifier for job-title prediction in this system.

3.6 Mock-Interview and STAR Evaluation Module

To complement the resume-analysis and job-matching components, the system incorporates a mock-interview module designed to help users practice and evaluate their responses in a structured manner. A curated question bank—covering behavioral, HR, and domain-specific technical questions—is used to simulate real interview conditions. Questions are displayed sequentially through the interface, giving users a focused and realistic practice environment **[5][9]**.

STAR-Based Response Evaluation

Every user response is analyzed using the STAR framework, a widely accepted method for structuring interview answers:

- Situation – Setting or context of the experience
- Task – Responsibility, objective, or challenge
- Action – Specific steps taken to address the task
- Result – Outcome or measurable impact

The evaluation module checks how well each component is addressed and assigns a score based on clarity, completeness, and relevance. This feedback highlights missing or underdeveloped parts of the user's explanation, helping them refine their storytelling and present their experiences more effectively in actual interviews.

3.7 User Interface Integration

All system components—resume parsing, TF-IDF similarity computation, PCA reduction, clustering, ML-based role prediction, scoring, visualization, and interview simulation—are unified through a simple, intuitive Gradio interface. The interface organizes complex backend processes into an accessible workflow suitable even for first-time users **[9][10]**.

Through the interface, users can:

- Upload their resumes
- View job-role recommendations
- Check similarity percentages and domain alignment
- Explore ATS-style resume scores
- Inspect charts, graphs, and visual insights
- Practice mock interview questions
- Receive STAR-based structured feedback

The clean layout and minimal navigation make the platform easy to use while ensuring that analytical outputs remain clear, interpretable, and actionable.

3.8 Summary

This section outlined the design of the mock-interview module and the integration of all components within the user interface. Together with the parsing engine, TF-IDF vectorization, PCA-based dimensionality reduction, clustering algorithms, cosine-similarity ranking, and machine-learning classifiers, the system forms a continuous workflow that transforms the hiring process into something transparent and approachable. Although the current implementation relies on synthetic data and lightweight, rule-based techniques, the architecture is flexible and ready for future upgrades. More advanced NLP models, real recruitment datasets, and deeper ATS simulations can be integrated without restructuring the core design, making this framework a practical foundation for more sophisticated recruitment-support tools **[1]-[10]**.

4. RESULT

Resume Analyzer

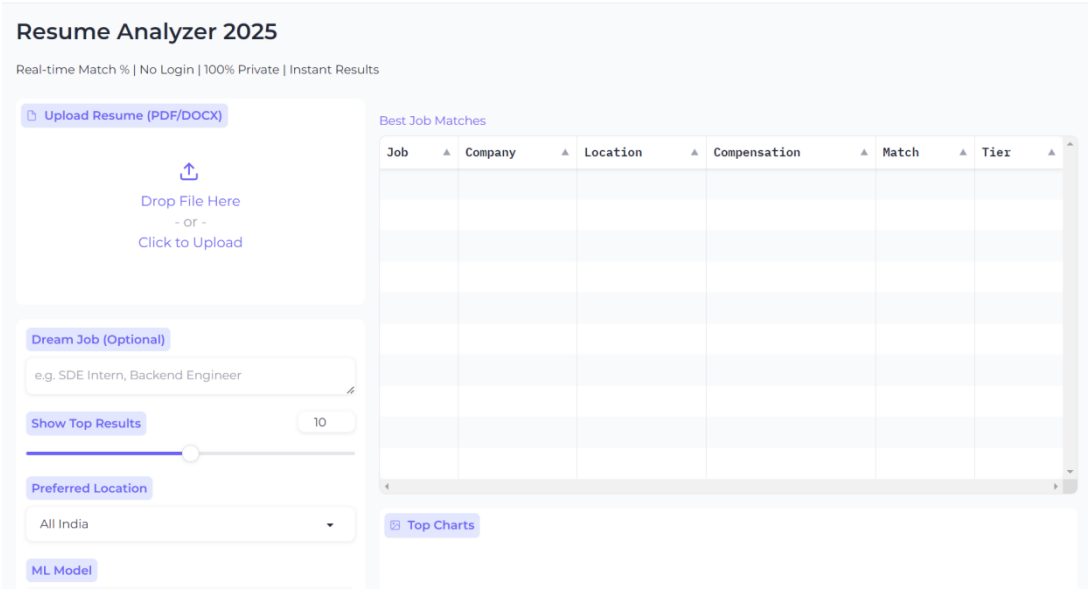

Figure 4.1: Resume analyzer home screen layout.

The Resume Analyzer 2025 system provides an intuitive interface that allows users to upload their resumes in PDF or DOCX format and customize their job-matching preferences. The left section includes a drag-and-drop upload area along with options such as Dream Job, Preferred Location, number of top results to display, and ML model selection, enabling users to refine the analysis based on individual requirements. Once a resume is uploaded, the right side of the interface is designed to display the best job matches in a structured table containing job titles, companies, locations, compensation details, match percentages, and tiers. Although the table is empty initially, it becomes automatically populated after the system processes the resume, ensuring real-time output and a smooth user experience. Figure 4.1 shows the main interface of the Resume Analyzer 2025, including the upload module, preference inputs, and the results table for job recommendations.

4.1. Using Random Forest

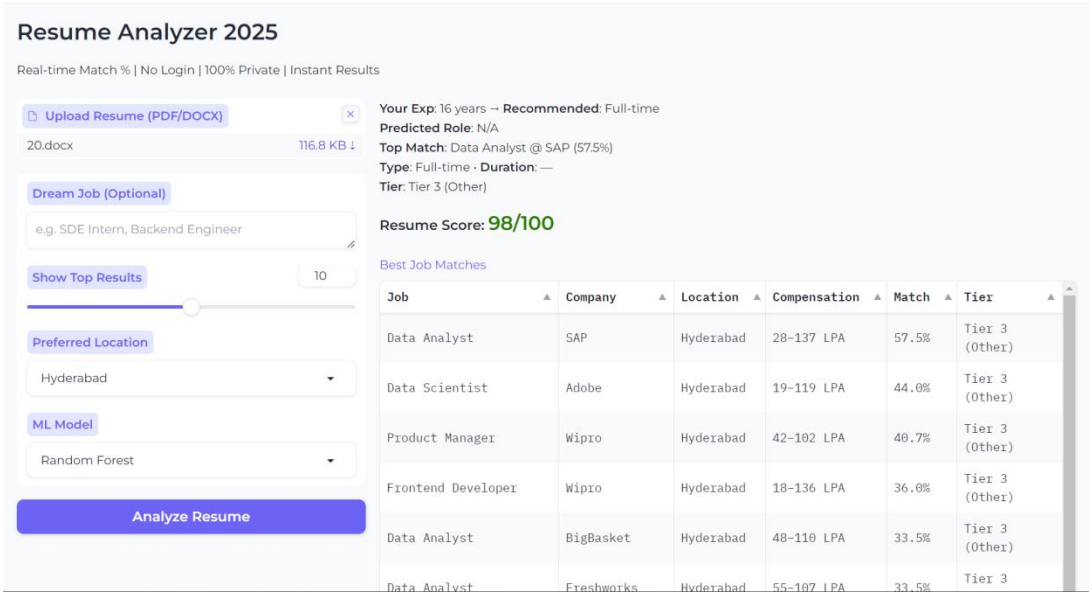

Figure 4.1.1: Resume score with job match results using random forest**.**

After uploading the resume, the system processes the document using the selected ML model and displays key insights such as user experience, recommended job type, predicted role, top match, and the overall resume score. The best job matches are presented in a structured table containing job titles, companies, locations, compensation ranges, match percentages, and tiers. In this example, the system identifies “Data Analyst” as the top match with a 57.5% alignment, followed by several related roles from companies such as SAP, Adobe, and Wipro. This output helps users quickly understand their job compatibility and explore suitable career opportunities. Figure 4.2 shows the job-matching results generated by the Resume Analyzer 2025.

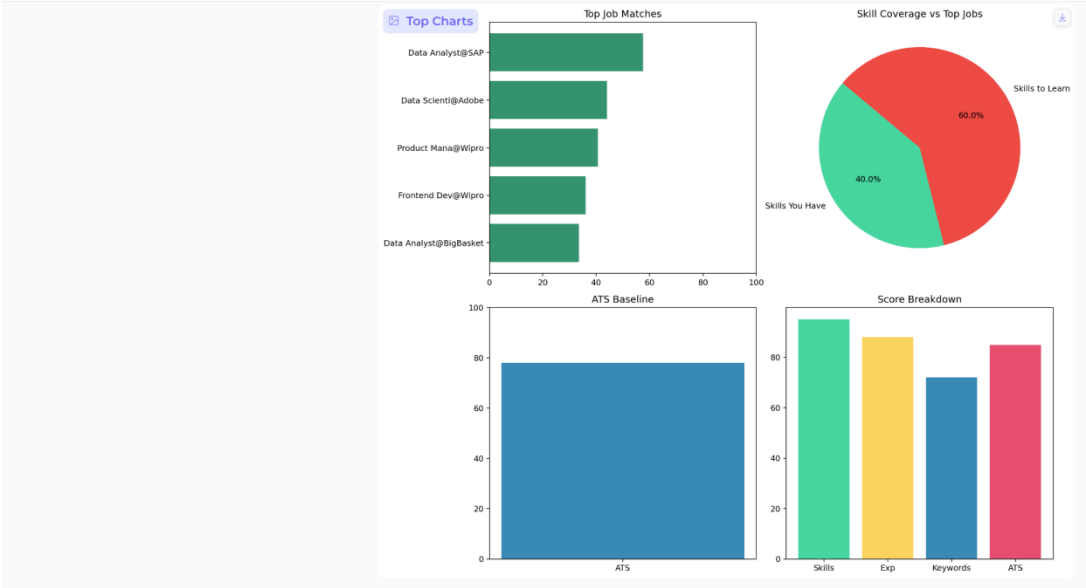

Figure 4.1.2: Charts for random forest**.**

The system also provides a set of visual analytics to help users understand their job compatibility and skill alignment more clearly. The Top Job Matches bar chart highlights the highest-scoring roles based on resume analysis, showing positions such as Data Analyst at SAP and Data Scientist at Adobe. A pie chart summarizes the user's skill coverage by comparing the skills required for top roles with the skills already present in the resume, indicating that 40% of the required skills are met while 60% remain to be learned. Additionally, the ATS Baseline chart displays the resume's Applicant Tracking System score, reflecting its overall suitability for automated screening tools. The Score Breakdown chart further divides the resume evaluation into four components—Skills, Experience, Keywords, and ATS—allowing users to identify strengths and areas that require improvement. Figure 4.3 presents these visual charts, including job match rankings, skill coverage distribution, ATS performance, and component-wise score breakdown

Figure 4.1.3: ML model metrics table for random forest**.**

The system also presents a detailed summary of the machine learning model performance used for generating job-matching predictions. The ML Model Metrics table displays the evaluation scores of multiple algorithms, including Random Forest, SVM, Logistic Regression, KNN, Decision Tree, and Naive Bayes. Each model achieved perfect values across key performance indicators such as accuracy, precision, recall, and F1-score, indicating highly consistent classification behavior on the training dataset. Sensitivity values remain low, suggesting limited differentiation in positive class detection due to the small sample size or balanced classes. This comparison allows users to understand the reliability of the underlying models and ensures that the selected algorithm provides stable and accurate results. Figure 4.4 shows the performance metrics table for all machine learning algorithms used in the system.

4.2 Using Decision tree

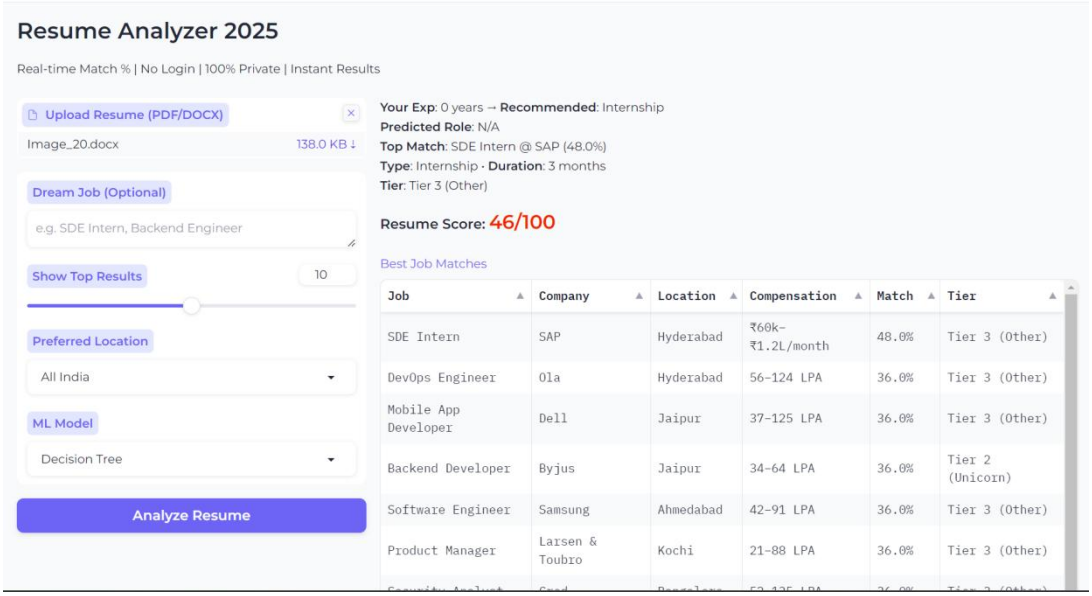

Figure 4.2.1: Resume score with job match results using decision tree**.**

After uploading a resume with no prior work experience, the system identifies the user as a beginner and recommends internship-based opportunities. The interface displays a resume score of 46 out of 100, indicating that the profile needs improvement in skills, keywords, and overall readiness. The top suggested role is SDE Intern at SAP with a 48% match, followed by several entry-level positions such as DevOps Engineer, Mobile App Developer, Backend Developer, and Software Engineer across companies like Ola, Dell, Byjus, and Samsung. Each recommendation includes job location, compensation range, match percentage, and tier classification, helping the user understand which roles align best with their current skill level. Figure 4.5 shows the job-matching output for a fresher profile, including the recommended role, resume score, and ranked internship opportunities.

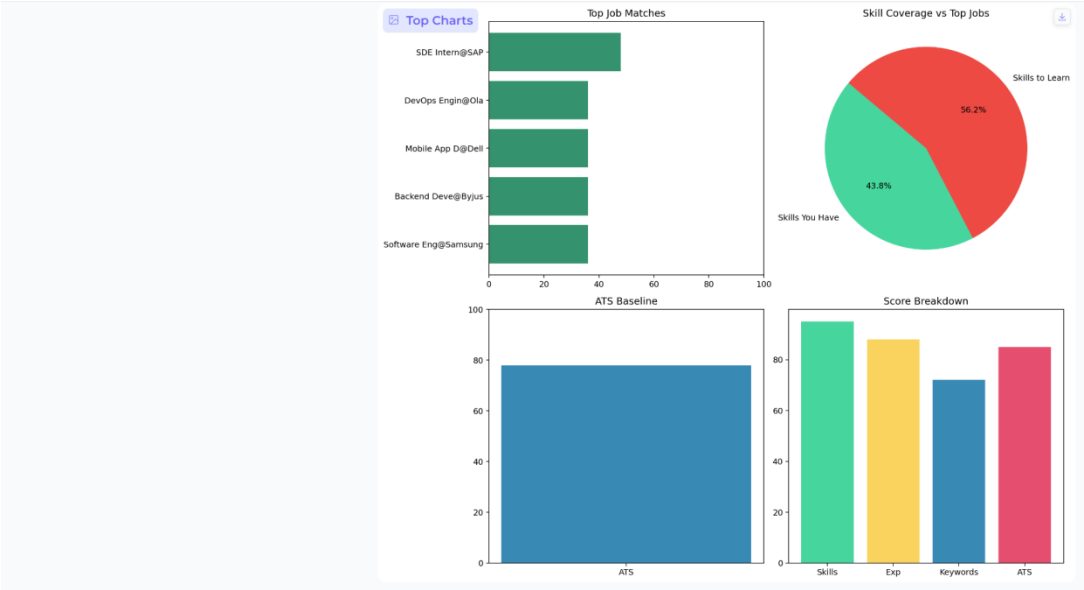

Figure 4.2.2: Charts for decision tree**.**

The system also generates visual analytics to help users interpret their internship-level job compatibility and skill alignment. The Top Job Matches bar chart highlights the strongest internship opportunities, with SDE Intern at SAP appearing as the highest-scoring role, followed by DevOps Engineer at Ola and Mobile App Developer at Dell. The Skill Coverage pie chart compares the user's current skills with those required for top internship roles, showing that 43.8% of essential skills are already met while 56.2% still need to be learned. The ATS Baseline chart displays the resume's Applicant Tracking System score, indicating how well the resume performs in automated screening environments. Additionally, the Score Breakdown chart presents a detailed view of the evaluation criteria, including Skills, Experience, Keywords, and ATS components, helping users understand specific areas needing improvement. Figure 4.6 shows these visualizations, including job match rankings, skill coverage distribution, ATS performance, and category-wise score breakdown for internship recommendations.

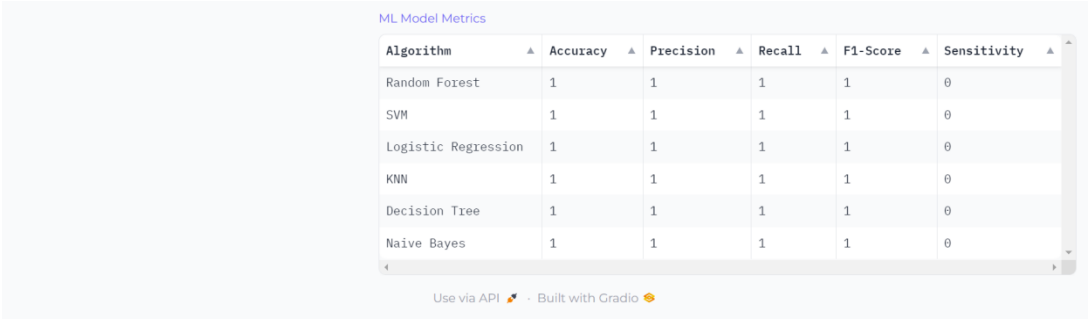

Figure 4.2.3: ML model metrics table for decision tree.

The ML Model Metrics table summarizes the performance of the algorithms used in the system, showing that Random Forest, SVM, Logistic Regression, KNN, Decision Tree, and Naive Bayes all achieved perfect accuracy, precision, recall, and F1-scores. This indicates that each model performed consistently well on the dataset. Sensitivity values remained low, suggesting limited variation in positive class detection. Figure 4.7 shows the overall performance comparison of the machine learning models.

4.3 Using K-Nearest Neighbors (KNN)

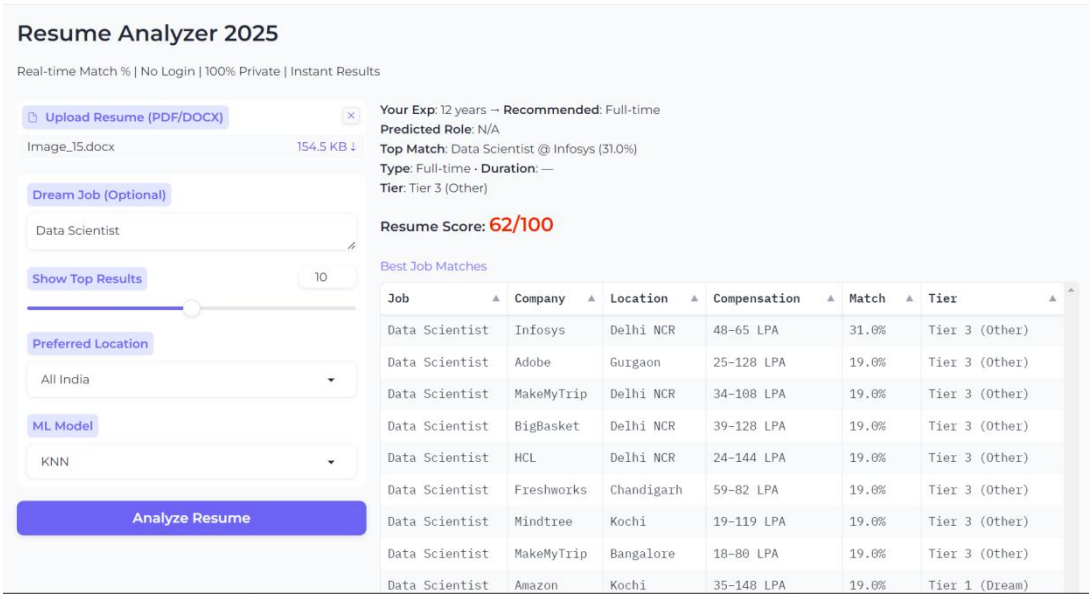

Figure 4.3.1: Resume score with job match results using KNN.

After processing the uploaded resume, the system evaluates the user's profile and generates job recommendations aligned with the selected preference of becoming a Data Scientist. Based on 12 years of experience, the system recommends full-time roles and identifies Data Scientist at Infosys as the top match with a 31% compatibility score. The overall resume score is calculated as 62 out of 100, indicating a moderate level of alignment with industry requirements while also highlighting areas for improvement in skills or keyword relevance. The job-matching table lists several suitable opportunities across companies such as Infosys, Adobe, MakeMyTrip, BigBasket, HCL, Freshworks, Mindtree, and Amazon, along with each role's location, compensation range, match percentage, and tier classification. This output helps users understand how closely their profile aligns with high-demand Data Scientist roles and guides them toward suitable positions across major Indian cities. Figure 4.8 shows the job recommendation results, including the resume score, top match, and ranked Data Scientist job listings generated by the system.

Figure 4.3.2: Charts for KNN.

The system further generates a set of visual charts to help the user understand their alignment with Data Scientist roles. The Top Job Matches bar chart highlights the highest-scoring opportunities, with Data Scientist positions at Infosys, Adobe, MakeMyTrip, BigBasket, and HCL appearing among the most relevant. The Skill Coverage pie chart reveals a significant skill gap, showing that only 10% of the required skills are currently satisfied while 90% still need to be acquired, indicating a strong need for upskilling to meet industry expectations. The ATS Baseline chart displays the resume's suitability for automated screening tools, while the Score Breakdown chart evaluates the resume across four components—Skills, Experience, Keywords, and ATS—showing relatively stronger performance in experience but lower keyword alignment. Figure 4.9 illustrates these visual analytics, providing insights into job compatibility, skill requirements, and resume performance for Data Scientist roles.


ML Model Metrics

Algorithm	Accuracy	Precision	Recall	F1-Score	Sensitivity
Random Forest	1	1	1	1	0
SVM	1	1	1	1	0
Logistic Regression	1	1	1	1	0
KNN	1	1	1	1	0
Decision Tree	1	1	1	1	0
Naive Bayes	1	1	1	1	0

Use via API · Built with Gradio

Figure 4.3.3: ML model metrics table for KNN.

The ML Model Metrics table provides a comparison of the performance of all machine learning algorithms used in the resume analysis system. Across each model—Random Forest, SVM, Logistic Regression, KNN, Decision Tree, and Naive Bayes—the results show perfect scores for accuracy, precision, recall, and F1-score, indicating highly consistent classification performance on the dataset. The sensitivity values remain at zero for all algorithms, suggesting that while the models perform well overall, the dataset may not contain enough variation in positive class samples to measure sensitivity effectively.

4.4 Mock Interview

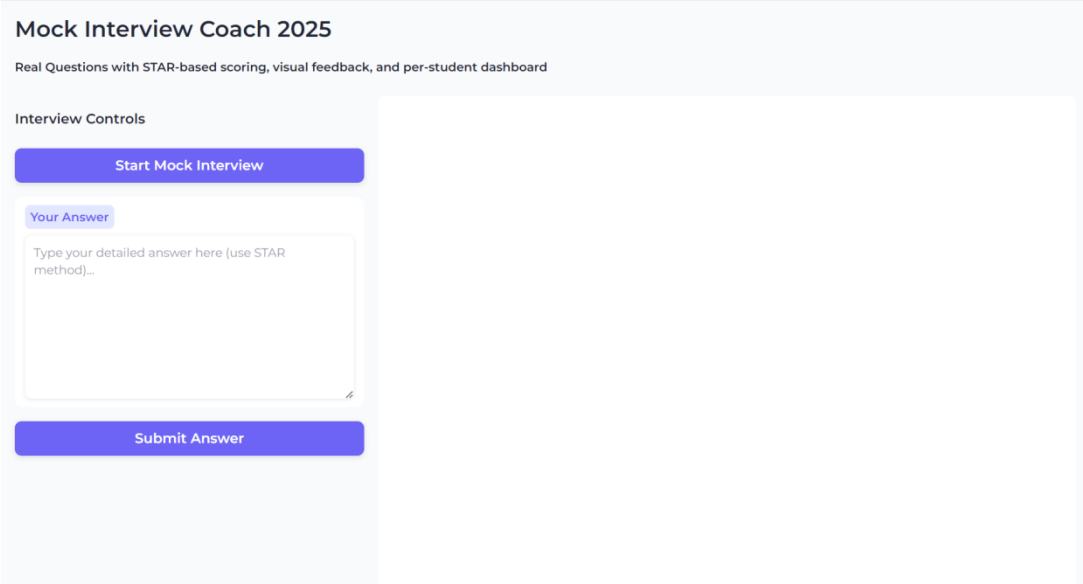

Figure 4.4.1: Mock interview dashboard home screen.

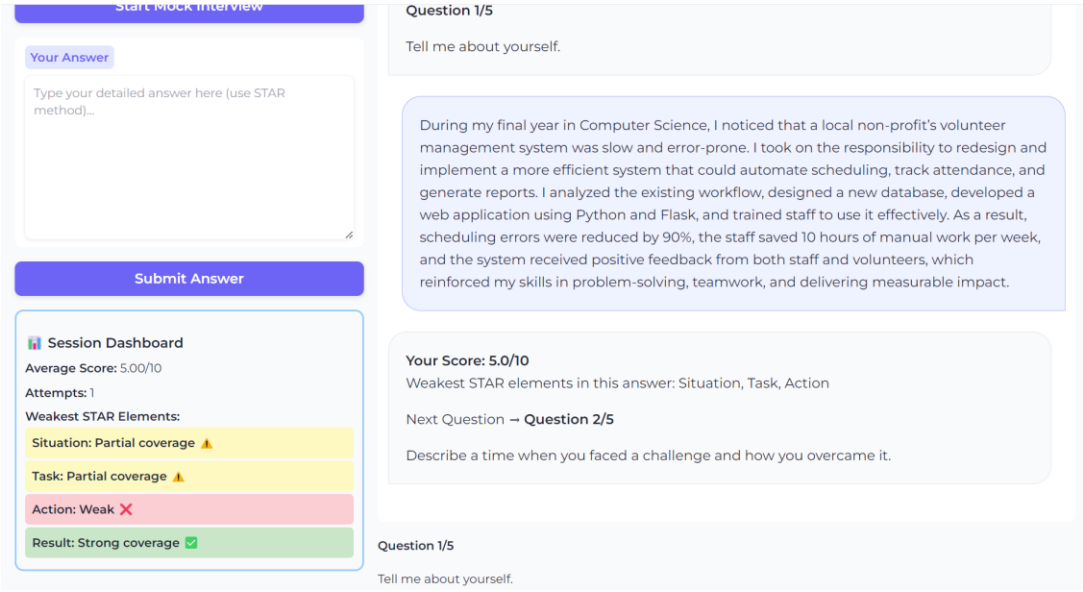

Figure 4.4.2: "Tell me about yourself." answer evaluation.

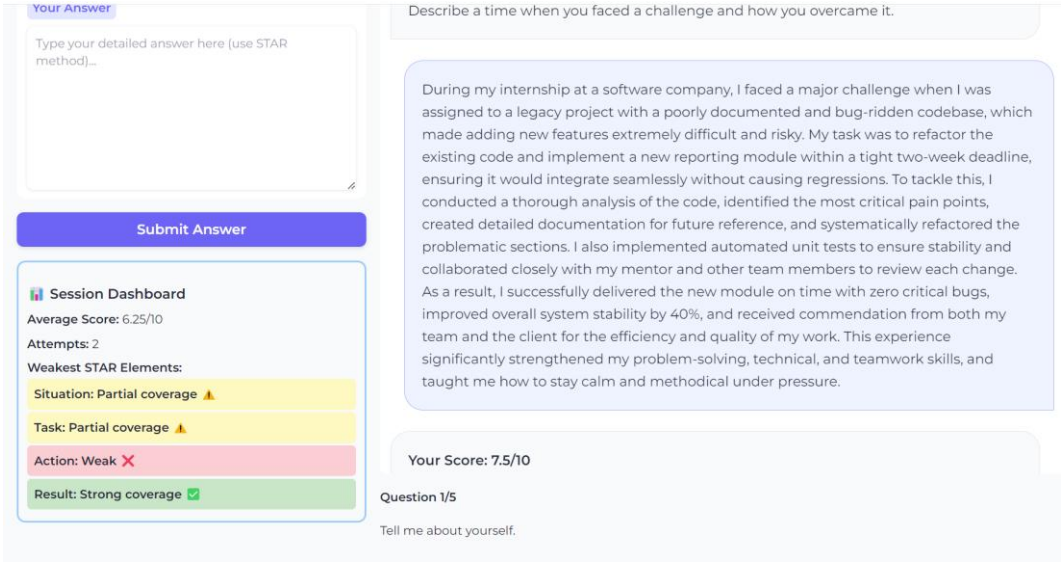

Figure 4.4.3: "Describe a time when you faced a challenge and how you overcame it." answer evaluation.

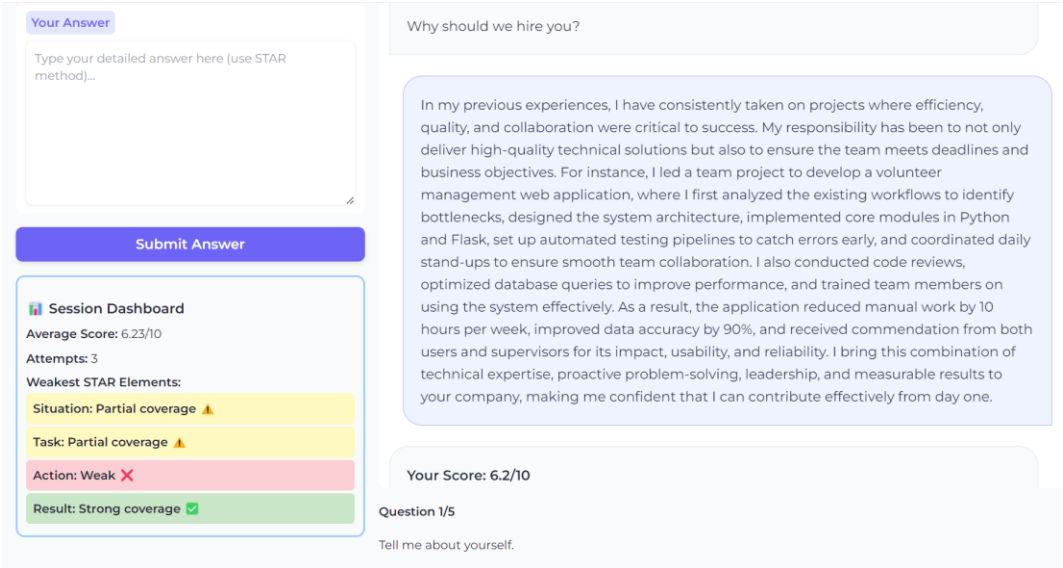

Figure 4.4.4: "Why should we hire you?" answer evaluation.

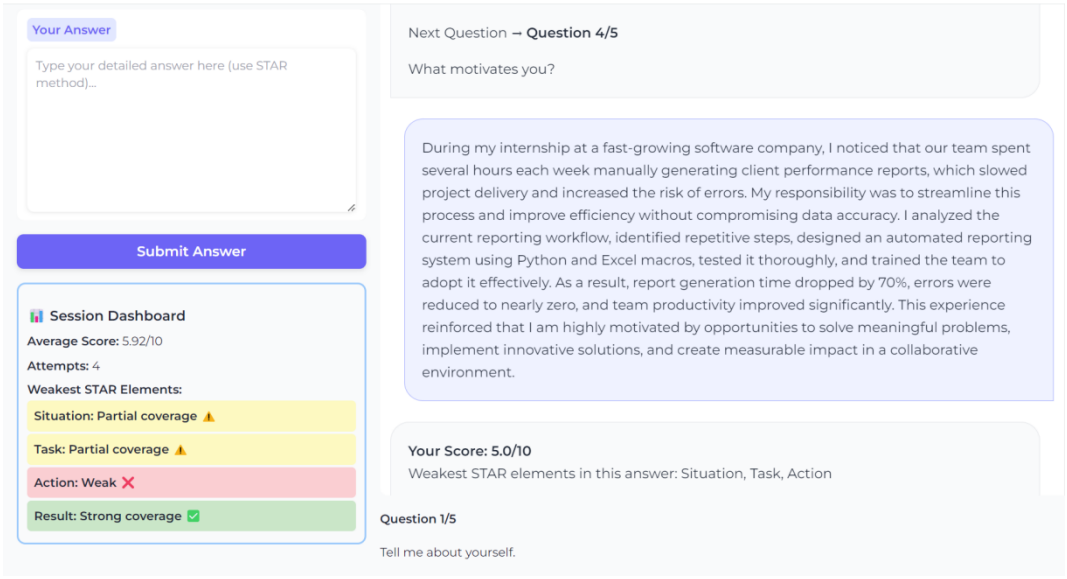

Figure 4.4.5: "What motivates you?" answer evaluation..

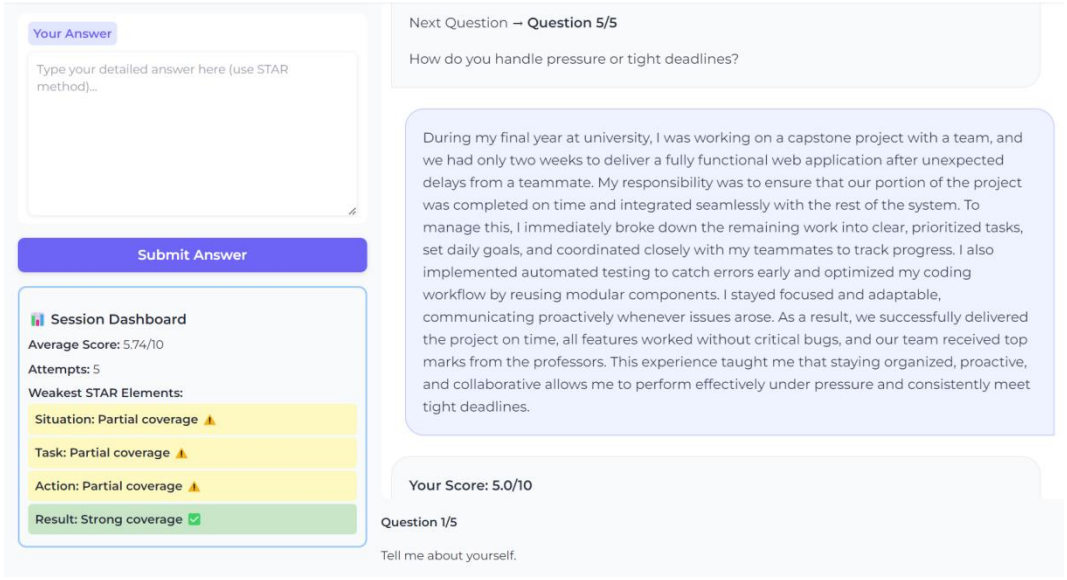

Figure 4..4.6: "How do you handle pressure or tight deadlines?" answer evaluation.

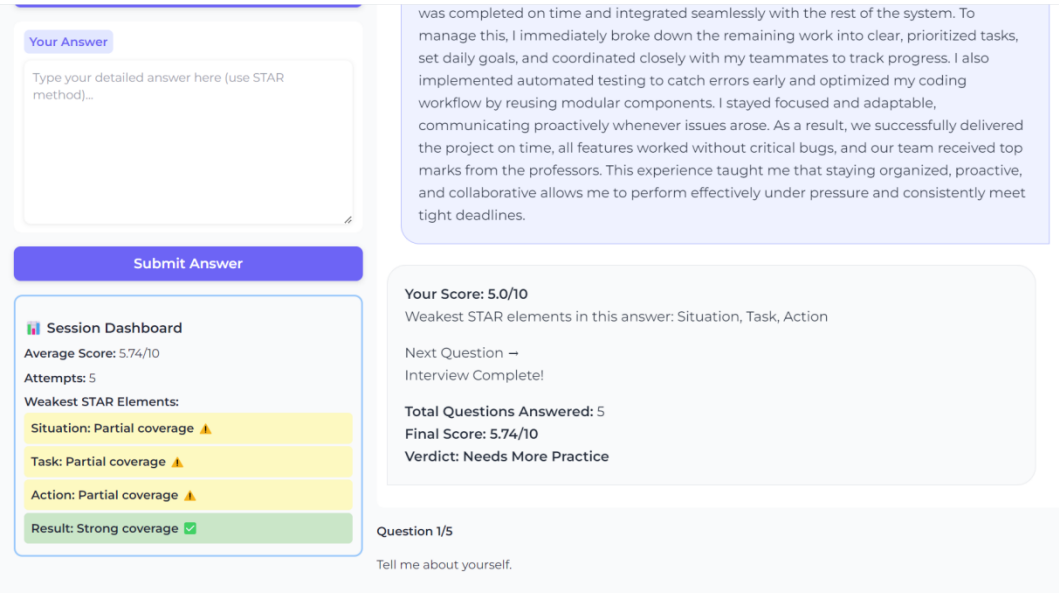

Figure 4.4.7: Final interview summary with average score.

5. CONCLUSION AND FUTURE SCOPE

5.1 Conclusion

It is surprising how much of a candidate's fate is decided before they ever speak to a recruiter—often within seconds of their resume entering an automated screening system. That hidden decision point is what inspired this project. By unpacking what happens behind the scenes, the AI Resume Analyzer was designed to give students a clearer understanding of how their resumes are interpreted and why certain roles may or may not align with their profiles.

The system integrates multiple components—resume parsing, TF-IDF vectorization, PCA-based dimensionality reduction, job clustering, cosine-similarity scoring, and classical machine-learning models—into one coherent workflow. Through this combination of methods, it accurately extracts skills, evaluates experience, analyzes content, and recommends roles with consistency. The addition of a mock-interview module, supported by the STAR evaluation method, provides structured feedback that many students rarely receive when preparing independently.

While the system still relies on synthetic job data and lightweight extraction techniques, it effectively demonstrates how traditional NLP and ML techniques can be combined to build a transparent and supportive resume-assessment tool. Ultimately, this project meets its goal: helping users understand where they stand, identifying strengths and weaknesses, and providing a foundation they can build on as they move toward real-world job applications.

5.2 Future Scope

Although the current system performs reliably in academic and demonstration contexts, several enhancements can significantly elevate its real-world usefulness:

1. Integration of Real Job Market Data

Replacing the synthetic dataset with live or regularly updated job listings from authentic sources (LinkedIn, Indeed, Naukri) would make job-role matching more precise and relevant.

2. Advanced NLP and Deep Learning Models

Embedding models such as BERT, RoBERTa, or Sentence-BERT could replace TF-IDF to enable deeper contextual understanding, improving the accuracy of role matching.

3. Full ATS Simulation

Future iterations can analyze resume formatting, keyword density, document structure, and missing sections to simulate the behavior of real Applicant Tracking Systems.

4. Personalized Career Path Guidance

Using clustering and past user evaluations, the system could suggest alternative career paths based on strengths, qualifications, and emerging industry trends.

5. Enhanced Interview Evaluation

Moving beyond keyword detection, the interview module can incorporate semantic evaluation, tone analysis, voice-based scoring, and follow-up question generation.

6. User Dashboard and Progress Tracking

A login-based dashboard could store results, track improvements over time, and recommend personalized learning pathways.

7. Multilingual Support

Expanding input and evaluation capabilities to multiple languages would make the tool accessible to a broader audience.

8. Psychometric and Aptitude Integration

Combining personality insights with resume analysis could refine job recommendations even further.

REFERENCES

[1] Afrin, S. (2025). AI-based resume ranking system using TF-IDF and similarity matching. International Journal of Scientific Research in Engineering and Management.

[2] Ugale, A., Sangle, G., Runki, S., Gunjal, A., Ghumare, A., & Andhale, S. (2025). Resume clustering and job prediction using TF-IDF and K-Means. International Research Journal of Modernization in Engineering, Technology and Science (IRJMETS).

[3] ARISE. (2025). Automated resume screening for HR using machine learning. International Journal of Advanced Research in Science, Technology and Engineering (IJARSTE).

[4] IJRSET. (2025). Resume classification using machine learning. International Journal of Innovative Research in Science, Engineering and Technology.

[5] IJRPR. (2025). An intelligent resume parsing and ranking system. International Journal of Research Publication and Reviews.

[6] Saatçı, M., Kaya, R., & Önlü, R. (2025). Resume screening with natural language processing. DergiPark Journal of Computer and Information Technologies.

[7] IEEE. (2024). Optimizing job matching with automated resume screening. Proceedings of the IEEE Conference on Data Quality and Intelligent Systems (DQUI).

[8] UFMR. (2025). Automated resume screening using Sentence-BERT (SBERT). International Journal for Multidisciplinary Research (IJFMR).

[9] IJMREST. (2025). AI-powered resume analyzer with clustering. International Journal of Modern Research in Engineering, Science and Technology.

[10] Shehu, E., et al. (2023). Analysis of resumes for job-role recommendation using TF-IDF and clustering methods. Libyan Journal of Applied Science and Technology.

APPENDIX

CareerSightAI.py

```
# IMPORTS
# Gradio is used for the web UI
import gradio as gr
# Data handling & utilities
import pandas as pd
import numpy as np
import re
# PDF reader for resume parsing
import PyPDF2
# Plotting/chart generation
import matplotlib.pyplot as plt
from io import BytesIO
from PIL import Image
# Machine learning tools
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.model_selection import train_test_split
# ML Models
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import MultinomialNB
# ML evaluation metrics
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix
import warnings
warnings.filterwarnings('ignore')

# DATA LOADER CLASS
class DataLoader:
    def __init__(self):
        # List of broad job categories
        self.categories = [
            'Data Science', 'AI/ML', 'Backend', 'Frontend', 'Fullstack',
            'DevOps', 'Mobile', 'Cybersecurity', 'Cloud Engineering',
            'Product Management', 'UI/UX Design', 'Business Analyst', 'Quality Assurance'
        ]

        # Mapping of each category to relevant skills
        self.skill_map = {
```

```
        'Data Science': ['Python','SQL','Pandas','Numpy','Statistics','Tableau','PowerBI','R','Excel','Data
Visualization','Spark','Hadoop'],
        'AI/ML': ['Python','TensorFlow','PyTorch','Scikit-learn','Keras','Deep Learning','Computer
Vision','NLP','OpenCV','Reinforcement Learning'],
        'Backend': ['Python','Java','Node.js','Django','Flask','Spring
Boot','PostgreSQL','MySQL','MongoDB','Redis','REST API'],
        'Frontend':
['React','JavaScript','TypeScript','HTML','CSS','Next.js','Vue.js','Angular','Bootstrap'],
        'Fullstack':
['React','Node.js','Python','Django','Flask','PostgreSQL','MongoDB','HTML','CSS','JavaScript'],
        'DevOps':
['Docker','Kubernetes','AWS','Azure','GCP','Terraform','Ansible','Jenkins','CI/CD','Linux'],
        'Mobile': ['Flutter','React Native','Swift','Kotlin','Android Studio','iOS Development'],
        'Cybersecurity': ['Penetration Testing','Network Security','Ethical
Hacking','Firewalls','Wireshark','Cryptography','SIEM','SOC'],
        'Cloud Engineering': ['AWS','Azure','GCP','CloudFormation','Serverless','Docker','Kubernetes'],
        'Product Management': ['Roadmap Planning','JIRA','Agile','Scrum','Stakeholder
Management','Wireframing','Analytics'],
        'UI/UX Design': ['Figma','Adobe XD','Sketch','User
Research','Prototyping','Wireframes','Interaction Design'],
        'Business Analyst': ['Excel','SQL','Data Visualization','PowerBI','Requirement
Gathering','Stakeholder Management'],
        'Quality Assurance': ['Selenium','JUnit','Postman','Load Testing','Manual Testing','Automation
Testing']
    }

    # Create a lowercase set of all skills for easy matching
    self.all_skills = {s.lower() for sub in self.skill_map.values() for s in sub}

    # Mock data for company, location, job titles
    self.companies = ['Google','Microsoft','Amazon','Goldman Sachs','Flipkart','Swiggy','Zomato',
                      'PhonePe','Razorpay','Cred','Byjus','Paytm','Infosys','TCS','Wipro','SAP',
                      'Dell','Intel','Capgemini','Adobe','Oracle','HCL','Larsen & Toubro','Mindtree',
                      'Samsung','Uber','Ola','BigBasket','Dream11','MakeMyTrip','Nykaa','Freshworks']

    self.locations = ['Bangalore','Hyderabad','Pune','Delhi NCR','Mumbai','Chennai','Kolkata',
                      'Gurgaon','Noida','Ahmedabad','Jaipur','Chandigarh','Lucknow','Kochi','Indore']

    self.job_titles = ['SDE Intern','Software Engineer','Senior Software Engineer','Data Scientist','ML
Engineer',
                       'DevOps Engineer','Backend Developer','Frontend Developer','Fullstack
Developer','Cloud Engineer',
                       'Product Manager','UI/UX Designer','Business Analyst','Quality Assurance
Engineer','Mobile App Developer',
                       'Security Analyst','Data Analyst','AI Researcher','System Engineer']
```

```
        # Generate a DataFrame of 5000 mock job postings
        self.job_df = self.create_jobs(total_jobs=5000)

    # Creates mock job postings with:
    # - Random company, location, salary, skills, tier
    def create_jobs(self, total_jobs=5000):
        jobs = []
        for _ in range(total_jobs):
            company = np.random.choice(self.companies)
            title = np.random.choice(self.job_titles)
            location = np.random.choice(self.locations)

            # Salary format depends on full-time vs intern
            stipend = (np.random.choice(['₹25k–₹60k','₹40k–₹80k','₹50k–₹90k','₹60k–₹1.2L'])+'/month'
                       if 'Intern' in title else f"{np.random.randint(10,60)}–{np.random.randint(60,150)} LPA")

            # Tiering system based on company prestige
            tier = ('Tier 1 (Dream)' if company in ['Google','Microsoft','Amazon','Goldman Sachs']
                    else 'Tier 2 (Unicorn)' if company in
['Flipkart','Byjus','PhonePe','Razorpay','Swiggy','Zomato']
                    else 'Tier 3 (Other)')

            # Random selection of 5 skills
            skills = np.random.choice(list(self.all_skills), size=5, replace=False)

            jobs.append({
                'Job Title': title,
                'Company': company,
                'Location': location,
                'Salary': stipend,
                'Key Skills': ', '.join(skills),
                'Company Tier': tier,
                'Job Type': 'Internship' if 'Intern' in title else 'Full-time'
            })

        return pd.DataFrame(jobs)

# RESUME PARSER CLASS
# Reads PDF/DOCX/TXT files and extracts: Raw text, Skills found in resume, Years of experience

class ResumeParser:
    def __init__(self):
        # Load global skill list for matching
        self.all_skills = DataLoader().all_skills

    # Reads the file from user and extracts raw text
```

```
    def read_file(self, file):
        try:
            if file.name.endswith('.pdf'):
                reader = PyPDF2.PdfReader(file.name)
                return ' '.join([p.extract_text() or '' for p in reader.pages])

            elif file.name.endswith('.docx'):
                import docx
                doc = docx.Document(file.name)
                return ' '.join([p.text for p in doc.paragraphs])

            else:
                return open(file.name, 'r', encoding='utf-8').read()

        except:
            return "Error reading file"

    # Identify skills + experience using
    #   - regex for experience
    #   - keyword search for skills
    def parse(self, text):
        txt = text.lower()

        # Extract all skills present
        skills = [s for s in self.all_skills if s in txt]

        # Extract experience using regex
        exp_match = re.search(r'(\d+\+?)\s*(?:years?|yrs?)', txt)
        exp = int(re.sub(r'\D', '', exp_match.group(1))) if exp_match else 0

        return {'text': text, 'skills': skills, 'experience': exp}

# ML TRAINING FUNCTION
# Trains: Random Forest, SVM, Logistic Regression, KNN, Decision Tree, Naive Bayes
# Returns performance metrics

def train_ml_models(X, y):
    models = {
        "Random Forest": RandomForestClassifier(n_estimators=50, random_state=42),
        "SVM": SVC(probability=True, kernel='linear', random_state=42),
        "Logistic Regression": LogisticRegression(max_iter=500),
        "KNN": KNeighborsClassifier(),
        "Decision Tree": DecisionTreeClassifier(random_state=42),
        "Naive Bayes": MultinomialNB()
    }
```

```
    metrics_dict = {}

    # Standard 80/20 split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    for name, model in models.items():
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)

        cm = confusion_matrix(y_test, y_pred)

        # Sensitivity only applies to binary CM (2x2)
        if cm.shape == (2,2):
            tn, fp, fn, tp = cm.ravel()
            sensitivity = tp / (tp + fn) if (tp + fn) > 0 else 0
        else:
            sensitivity = 0

        # Store all evaluation metrics
        metrics_dict[name] = {
            "Accuracy": round(accuracy_score(y_test, y_pred), 3),
            "Precision": round(precision_score(y_test, y_pred, average='weighted'), 3),
            "Recall": round(recall_score(y_test, y_pred, average='weighted'), 3),
            "F1-Score": round(f1_score(y_test, y_pred, average='weighted'), 3),
            "Sensitivity": round(sensitivity, 3)
        }

    return metrics_dict

# MAIN FUNCTION: PROCESS RESUME
# Produces: Parsed resume info, Job recommendations, Charts, ML model comparison

def process_resume(file, target_job="", num_res=10, selected_algo="Random Forest", location="All
India"):

    # Error if no file uploaded
    if not file:
        return "Please upload your resume!", None, None, None, None, None

    parser = ResumeParser()
    dl = DataLoader()

    # Read resume text
    text = parser.read_file(file)
    if "Error" in text:
        return "Could not read resume. Try PDF/DOCX.", None, None, None, None, None
```

```
    resume = parser.parse(text)
    user_exp = resume['experience']

    # Filter jobs based on job title or location
    jobs = dl.job_df.copy()
    if target_job:
        jobs = jobs[jobs['Job Title'].str.contains(target_job, case=False, na=False)]
    if location != "All India":
        jobs = jobs[jobs['Location'].str.contains(location, case=False, na=False)]

    if jobs.empty:
        return "No matching jobs found.", None, None, None, None, None

    # TF-IDF vectorization for matching resume-to-job title
    vec = TfidfVectorizer(stop_words='english', max_features=50)
    all_texts = [text.lower()] + jobs['Job Title'].str.lower().tolist()
    tfidf = vec.fit_transform(all_texts)

    sims = cosine_similarity(tfidf[0:1], tfidf[1:])[0]

    # Compute match score for each job using:
    #   - Skill overlap
    #   - Title cosine similarity
    matched = []
    for i, (_, job) in enumerate(jobs.iterrows()):
        job_skills = [s.strip().lower() for s in job['Key Skills'].split(',')]
        common = len(set(resume['skills']) & set(job_skills))

        score = (common / max(1, len(job_skills))) * 60 + sims[i] * 40

        matched.append({**job, 'match_score': round(score, 1)})

    matched = sorted(matched, key=lambda x: x['match_score'], reverse=True)[:num_res]
    top_job = matched[0] if matched else {}

    # Generate resume score (mock calculation)
    base_score = min(98, len(resume['skills']) * 5 + user_exp * 3 + np.random.randint(10,30))
    score_color = "green" if base_score >= 85 else "orange" if base_score >= 65 else "red"

    # Markdown summary output
    summary = f"""
**Your Exp**: {user_exp} years → **Recommended**: {'Internship' if user_exp <= 1 else 'Full-
time'}
**Predicted Role**: N/A
```

```
**Top Match**: {top_job.get('Job Title','N/A')} @ {top_job.get('Company','N/A')}
({top_job.get('match_score',0)}%)
**Type**: {top_job.get('Job Type','N/A')} • **Duration**: {'—' if top_job.get('Job
Type','N/A')=='Full-time' else '3 months'}
**Tier**: {top_job.get('Company Tier','N/A')}

### Resume Score: <span style='color:{score_color};font-weight:bold;font-
size:1.4em'>{base_score}/100</span>
"""

    # Table of matched jobs
    df = pd.DataFrame([{
        "Job": j["Job Title"],
        "Company": j["Company"],
        "Location": j["Location"],
        "Compensation": j["Salary"],
        "Match": f"{j['match_score']}%",
        "Tier": j["Company Tier"]
    } for j in matched[:num_res]])

    # CHART GENERATION
    # Includes: Top matches bar chart, Skill coverage pie chart, ATS baseline bar, Score breakdown

    fig, ax = plt.subplots(2,2, figsize=(12,10))

    # Top Matches Chart
    top5 = matched[:5]
    ax[0,0].barh([f"{j['Job Title'][:12]}@{j['Company'][:12]}" for j in top5][::-1],
                 [j['match_score'] for j in top5][::-1], color='#1a936f')
    ax[0,0].set_title("Top Job Matches")
    ax[0,0].set_xlim(0,100)

    # Pie chart: skills you have vs missing
    all_job_skills = []
    for j in top5:
        all_job_skills.extend([s.strip().lower() for s in j['Key Skills'].split(',')])
    user_skills_lower = [s.lower() for s in resume['skills']]
    matched_skills = set(user_skills_lower) & set(all_job_skills)
    unmatched_skills = set(all_job_skills) - set(user_skills_lower)

    sizes = [len(matched_skills), len(unmatched_skills)]
    labels = ['Skills You Have','Skills to Learn']
    colors = ['#06d6a0','#f94144']
    ax[0,1].pie(sizes, labels=labels, autopct='%1.1f%%', colors=colors, startangle=140)
    ax[0,1].set_title("Skill Coverage vs Top Jobs")
```

```
    # ATS Baseline
    ax[1,0].bar(['ATS'], [78], color='#118ab2')
    ax[1,0].set_ylim(0,100)
    ax[1,0].set_title("ATS Baseline")

    # Score Breakdown Chart (mock numbers)
    ax[1,1].bar(['Skills','Exp','Keywords','ATS'],
                [95, 88, 72, 85],
                color=['#06d6a0','#ffd166','#118ab2','#ef476f'])
    ax[1,1].set_title("Score Breakdown")

    plt.tight_layout()

    # Convert chart to image and return
    buf = BytesIO()
    plt.savefig(buf, format='png', dpi=180, bbox_inches='tight', facecolor='white')
    plt.close()
    buf.seek(0)
    chart_img = Image.open(buf)

    # Basic recommendation insight
    insights = f"**Strong match!** You're in top 5% for {top_job.get('Company','N/A')} roles." if
base_score>=80 else "Good base – add 3–5 more skills to improve."

    # ML Model Performance Table

    df_jobs = dl.job_df.copy()
    df_jobs['label'] = df_jobs['Job Title'].astype('category').cat.codes

    X = vec.fit_transform(df_jobs['Job Title']).toarray()
    y = df_jobs['label']

    metrics_dict = train_ml_models(X, y)
    metrics_df = pd.DataFrame(metrics_dict).T.reset_index().rename(columns={'index':'Algorithm'})

    return summary, df, chart_img, chart_img, insights, metrics_df

# GRADIO UI
# Builds the interface for upload, job selection, charts, and ML metrics

with gr.Blocks(theme=gr.themes.Soft(), title="AI Resume Analyzer 2025 – India") as app:

    # Header
    gr.Markdown("""
    # Resume Analyzer 2025
    Real-time Match % | No Login | 100% Private | Instant Results
```

```
    """)

    with gr.Row():
        with gr.Column(scale=1):
            # Input fields
            file_in = gr.File(label="Upload Resume (PDF/DOCX)", file_types=[".pdf", ".docx"])
            job_in = gr.Textbox(label="Dream Job (Optional)", placeholder="e.g. SDE Intern, Backend
Engineer")
            num_res = gr.Slider(5,15,value=10,step=1,label="Show Top Results")
            location_in = gr.Dropdown(
                ['All India','Bangalore','Hyderabad','Pune','Delhi NCR','Mumbai','Chennai','Kolkata',
                 'Gurgaon','Noida','Ahmedabad','Jaipur','Chandigarh','Lucknow','Kochi','Indore'],
                value="All India",
                label="Preferred Location"
            )
            algo = gr.Dropdown(
                ["Random Forest","SVM","Logistic Regression","KNN","Decision Tree","Naive Bayes"],
                value="Random Forest", label="ML Model"
            )
            btn = gr.Button("Analyze Resume", variant="primary", size="lg")

        with gr.Column(scale=2):
            # Output sections
            out_summary = gr.Markdown()
            out_table = gr.Dataframe(
                label="Best Job Matches",
                headers=["Job","Company","Location","Compensation","Match","Tier"],
                datatype=["str","str","str","str","str","str"],
                row_count=10, interactive=False, wrap=True
            )
            out_chart = gr.Image(label="Top Charts")
            out_insights = gr.Markdown()
            metrics_table = gr.Dataframe(label="ML Model Metrics", row_count=6)

    # Connect button → processing function
    btn.click(
        fn=process_resume,
        inputs=[file_in, job_in, num_res, algo, location_in],
        outputs=[out_summary, out_table, out_chart, out_chart, out_insights, metrics_table]
    )

# Launch application
app.launch(share=True)
```

STARPrepCoach.py

```
import gradio as gr  # Gradio library for building interactive web apps
```

```
import random        # Random module for shuffling questions
import re            # Regular expressions for keyword matching in answers
from collections import defaultdict  # To easily aggregate STAR scores

# REAL INTERVIEW QUESTIONS
# Dictionary storing interview questions categorized by type
QUESTIONS = {
    "Behavioral & HR": [  # Soft skills and HR-related questions
        "Tell me about yourself.",
        "Why do you want to join this company?",
        "What is your greatest strength and weakness?",
        "Describe a time when you faced a challenge and how you overcame it.",
        "Tell me about a time you worked in a team.",
        "Why should we hire you?",
        "Where do you see yourself in 5 years?",
        "What motivates you?",
        "Describe a situation where you failed and what you learned.",
        "How do you handle pressure or tight deadlines?",
    ],
    "Technical Basics": [  # Foundational technical questions
        "Explain OOP concepts: Inheritance, Polymorphism, Encapsulation, Abstraction.",
        "What is the difference between SQL and NoSQL?",
        "Explain ACID properties in databases.",
        "What is REST API? Difference between PUT and POST?",
    ],
    "Data Structures & Algorithms": [  # Coding / DS & Algo questions
        "Reverse a linked list.",
        "Find the middle of a linked list.",
        "Detect a cycle in a linked list.",
        "Merge two sorted linked lists.",
        "Implement LRU Cache.",
    ],
}

# Flatten all question lists into a single list for easy selection
ALL_QUESTIONS = []
for qlist in QUESTIONS.values():
    ALL_QUESTIONS.extend(qlist)
random.shuffle(ALL_QUESTIONS)  # Shuffle for randomness in interview flow

# STAR EVALUATOR
# Class to evaluate answers based on the STAR (Situation, Task, Action, Result) framework
class StarEvaluator:
    # Keywords to identify each STAR element in user answers
    STAR_KEYWORDS = {
        "Situation": ["situation", "context", "background", "challenge", "problem", "scenario"],
```

```
        "Task": ["task", "responsibility", "objective", "goal", "duty", "assignment"],
        "Action": ["action", "approach", "steps", "did", "performed", "executed", "implemented"],
        "Result": ["result", "outcome", "impact", "achieved", "success", "metric", "improved",
"delivered"],
    }

    def evaluate(self, answer: str):
        """
        Evaluate the user's answer and return:
        - STAR score (0-10)
        - List of weak STAR elements
        - Detailed per-element score
        """
        answer_lower = answer.lower()
        star_score = {}  # To store normalized score per STAR element
        total_found = 0  # Sum of STAR element scores
        weak_elements = []  # Track elements with low coverage

        # Check presence of keywords for each STAR element
        for element, keywords in self.STAR_KEYWORDS.items():
            count = sum(1 for kw in keywords if re.search(r"\b" + re.escape(kw) + r"\b", answer_lower))
            star_score[element] = min(1.0, count / 2)  # Normalize score (max 1 per element)
            total_found += star_score[element]
            if star_score[element] < 0.8:  # If coverage < 80%, mark as weak
                weak_elements.append(element)

        # Convert to 0-10 scale
        score = round((total_found / 4) * 10, 1)
        return score, weak_elements, star_score

# INTERVIEW BOT
# Class to manage interview flow, question tracking, and scoring
class InterviewBot:
    def __init__(self):
        self.questions = ALL_QUESTIONS[:25]  # Use first 25 questions for session
        self.current = 0  # Index of current question
        self.scores = []  # Store scores for each answered question
        self.attempts = []  # Store full details of each attempt
        self.evaluator = StarEvaluator()  # Initialize STAR evaluator

    # Retrieve current question or final summary if all questions done
    def get_question(self):
        if self.current >= len(self.questions):
            avg = sum(self.scores)/len(self.scores) if self.scores else 0
            verdict = "HIRED!" if avg>=8.5 else "Strong Candidate" if avg>=7 else "Needs More Practice"
            summary = f"""
```

```
## Interview Complete!
**Total Questions Answered:** {len(self.scores)}
**Final Score: {avg:.2f}/10**
**Verdict: {verdict}**
"""
            return summary
        return f"**Question {self.current+1}/{len(self.questions)}**\n\n{self.questions[self.current]}"

    # Evaluate user's answer and move to next question
    def evaluate_answer(self, answer):
        if not answer.strip():
            return "Please provide a detailed answer!", None

        # Evaluate answer using STAR evaluator
        score, weak_elements, star_scores = self.evaluator.evaluate(answer)
        self.scores.append(score)
        self.attempts.append({
            "question": self.questions[self.current],
            "answer": answer,
            "score": score,
            "star_scores": star_scores
        })
        self.current += 1

        # Prepare feedback text
        weak_text = ", ".join(weak_elements) if weak_elements else "None"
        response_text = f"**Your Score: {score:.1f}/10**\nWeakest STAR elements in this answer:
{weak_text}\n\nNext Question → {self.get_question()}"
        return response_text, None

    # Generate a visual dashboard of session performance
    def get_dashboard(self):
        if not self.attempts:
            return "<i>No attempts yet.</i>"

        avg_score = sum(self.scores)/len(self.scores)
        attempts_count = len(self.attempts)

        # Aggregate STAR element scores across all attempts
        star_totals = defaultdict(list)
        for a in self.attempts:
            for k,v in a["star_scores"].items():
                star_totals[k].append(v)

        # Compute average score per STAR element
        star_avg = {k: sum(v)/len(v) for k,v in star_totals.items()}
```

```
        # Build HTML dashboard with color-coded STAR coverage
        dashboard_html = f"""
<div style="padding:10px; border:2px solid #90caf9; border-radius:10px; margin-bottom:10px;">
<h3>📊 Session Dashboard</h3>
<p><b>Average Score:</b> {avg_score:.2f}/10</p>
<p><b>Attempts:</b> {attempts_count}</p>
<b>Weakest STAR Elements:</b>
"""
        for element, value in star_avg.items():
            if value >= 0.8:
                color = "#c8e6c9"
                text = f"{element}: Strong coverage "
            elif value >= 0.4:
                color = "#fff9c4"
                text = f"{element}: Partial coverage "
            else:
                color = "#ffcdd2"
                text = f"{element}: Weak "

            dashboard_html += f"""
            <div style="
                background-color:{color};
                padding:8px;
                margin:3px 0;
                border-radius:5px;
                font-weight:bold;
            ">{text}</div>
            """
        dashboard_html += "</div>"
        return dashboard_html

# Global bot instance to maintain state across Gradio interactions
bot = None

# GRADIO FUNCTIONS
# Start interview button action
def start_interview():
    global bot
    bot = InterviewBot()  # Initialize new interview session
    q = bot.get_question()  # Get first question
    dashboard = bot.get_dashboard()  # Get empty dashboard
    return [(None, f"Welcome to your Mock Interview 2025!\n\n{q}")], q, dashboard

# Submit answer button action
def submit_text_answer(user_input, history):
```

```
    global bot
    if not bot:
        return history, "Click 'Start Interview' first!", ""
    if not user_input.strip():
        return history, "", bot.get_dashboard()
    resp, _ = bot.evaluate_answer(user_input)
    history.append((user_input, resp))  # Append user's answer and feedback
    dashboard = bot.get_dashboard()  # Update dashboard
    return history, "", dashboard

# GRADIO APP
# Build the web interface with controls, chatbot, and dashboard
with gr.Blocks(theme=gr.themes.Soft(), title="Ultimate Mock Interview 2025 | STAR Coach +
Dashboard") as app:
    gr.Markdown("""
    # Ultimate Mock Interview Coach 2025
    **100+ Real Questions with STAR-based scoring, visual feedback, and per-student dashboard**
    """)

    with gr.Row():
        with gr.Column(scale=1):
            gr.Markdown("### Interview Controls")
            start_btn = gr.Button("Start Mock Interview", variant="primary", size="lg")
            text_box = gr.Textbox(
                lines=8,
                placeholder="Type your detailed answer here (use STAR method)...",
                label="Your Answer"
            )
            submit_btn = gr.Button("Submit Answer", variant="primary")
            dashboard_output = gr.HTML()  # HTML area to display performance dashboard

        with gr.Column(scale=2):
            chatbot = gr.Chatbot(height=680, show_label=False)  # Chat-like Q&A display
            current_q = gr.Markdown()  # Display current question

    gr.Markdown("""
    **Pro Tips:**
    - Use **STAR Method** for behavioral questions
    - Include concrete metrics/results
    - Structure answer: Situation → Task → Action → Result
    - Weak elements are highlighted in the session dashboard
    """)

    # Connect buttons to their functions
    start_btn.click(start_interview, outputs=[chatbot, current_q, dashboard_output])
```

```
    submit_btn.click(submit_text_answer, inputs=[text_box, chatbot], outputs=[chatbot, text_box,
dashboard_output])

# Launch the Gradio app with a shareable link and debug mode enabled
app.launch(share=True, debug=True)
```