AI Resume Analyzer & Job Match Platform

Project Report

Sai Varshini Papineni

July 21, 2025

Table of Contents

1. Abstract / Executive Summary

This report details the development of an "AI Resume Analyzer & Job Match Platform," an intelligent system designed to automate and enhance the process of matching resumes with job descriptions. The platform addresses the common challenges faced by both job seekers (optimizing resumes for specific roles) and recruiters (efficiently screening large volumes of applications). By integrating Natural Language Processing (NLP) techniques with a locally hosted Large Language Model (LLM) via Ollama, the system provides a quantitative match score and qualitative, AI-generated feedback. Key functionalities include robust text extraction from various document formats (PDF, DOCX), advanced text preprocessing using spaCy, cosine similarity for relevance scoring, and a user-friendly web interface built with Streamlit. The project emphasizes data privacy by performing all sensitive AI inference locally. Initial testing demonstrates the platform's ability to provide relevant insights, significantly reducing manual screening time and offering actionable advice for resume tailoring. Challenges encountered, such as environment management and refining LLM output, were addressed, paving the way for future enhancements to improve accuracy and user experience.

2. Introduction

The modern job market is characterized by a high volume of applications per job opening, making the manual screening of resumes a time-consuming and often inefficient process for recruiters. Simultaneously, job seekers face the challenge of tailoring their resumes to bypass Applicant Tracking Systems (ATS) and stand out to hiring managers. This project aims to bridge this gap by developing an AI-powered platform that intelligently analyzes resumes and job descriptions, providing data-driven insights for both parties.

2.1. Problem Statement

Current resume screening methods are often manual, prone to human bias, and time-intensive. For job seekers, understanding how well their resume aligns with a specific job description can be opaque, leading to generic applications that may be overlooked. For recruiters, the sheer volume of applications necessitates automated tools, but many existing solutions lack the nuance and contextual understanding required for effective matching and personalized feedback. The reliance on cloud-based AI solutions also raises concerns about data privacy for sensitive personal information contained in resumes.

2.2. Project Objectives

The primary objectives of this project are:

- To design and implement an AI-powered platform capable of performing intelligent resume analysis.
- To develop a robust mechanism for accurately comparing resume content with job descriptions using advanced NLP techniques.
- To provide a quantitative match score (e.g., percentage) indicating the relevance of a resume to a job description.
- To generate qualitative, prompt-based feedback from a Large Language Model (LLM) offering tailored suggestions for resume improvement.
- To build a user-friendly web-based interface for easy interaction, accessible via standard web browsers.
- To ensure that the core AI processing is performed locally using free and open-source tools (Ollama), addressing data privacy concerns.
- To document the project thoroughly, including setup, usage, and code, for reproducibility and future development.

2.3. Scope of the Project

This project focuses on the core functionality of resume analysis and job matching. It includes:

- Text extraction from PDF and DOCX resume formats.
- NLP-based preprocessing of both resume and job description texts.
- Calculation of cosine similarity as a quantitative match metric.
- Integration with a local LLM (Gemma via Ollama) for generating qualitative, personalized feedback.
- A Streamlit-based web user interface for input (file upload, text area) and output display.
- Exclusion from scope: User authentication, database storage of resumes/job descriptions, real-time job board scraping, advanced analytics dashboards, or deep neural network training from scratch. These are considered future enhancements.

3. Background and Literature Review

The development of this platform draws upon several key areas of artificial intelligence and software engineering.

3.1. Natural Language Processing (NLP)

NLP is a field of artificial intelligence that enables computers to understand, interpret, and generate human language. In this project, NLP is fundamental for:

- **Text Extraction:** Converting unstructured document formats (PDF, DOCX) into raw, readable text.
- **Text Preprocessing:** Cleaning and normalizing text data by removing noise (e.g., punctuation, special characters), converting to lowercase, tokenizing (breaking text into words), and lemmatizing (reducing words to their base form). This prepares the text for accurate analysis.
- **Feature Extraction:** Transforming text into numerical representations (vectors) that machine learning models can understand. TF-IDF (Term Frequency-Inverse Document Frequency) is a common technique used here, which weighs words based on their importance in a document relative to a collection of documents.

We specifically use spaCy, a highly efficient and industrial-strength NLP library for Python, for its robust text processing capabilities and pre-trained language models.

3.2. Large Language Models (LLMs) and Ollama

Large Language Models (LLMs) are deep learning models trained on vast amounts of text data, enabling them to understand, generate, and answer questions in human-like language. While many powerful LLMs are cloud-based (e.g., OpenAI's GPT, Google's Gemini), this project prioritizes privacy and local execution.

Ollama is an open-source framework that allows users to download, run, and manage LLMs directly on their local machines. It provides a simple command-line interface and an API for interacting with these models. By using Ollama, we can leverage the power of LLMs like Gemma (a lightweight yet capable model from Google) for nuanced resume feedback without sending sensitive data to external servers. This setup ensures data remains on the user's computer, addressing critical privacy concerns in resume analysis.

3.3. Resume Parsing Techniques

Resume parsing involves extracting structured information (e.g., name, contact, skills, experience, education) from unstructured resume documents. This project employs two popular Python libraries for this:

- **pypdf (formerly PyPDF2):** A pure-Python PDF library capable of reading and extracting text from PDF documents.
- **python-docx:** A library for creating, reading, and updating Microsoft Word (.docx) files.

While these libraries provide raw text extraction, advanced parsing to structure specific fields (like "Experience" or "Skills" into distinct data points) would require more complex NLP techniques or rule-based systems, which are partially handled by the LLM's understanding in this project.

3.4. Cosine Similarity for Text Matching

Cosine similarity is a metric used to measure how similar two non-zero vectors are. In the context of text analysis, after converting documents into numerical vectors (e.g., using TF-IDF), cosine similarity calculates the cosine of the angle between these vectors. A value close to 1 indicates high similarity (small angle), while a value close to 0 indicates low similarity (large angle). This mathematical approach allows for a quantitative assessment of how well a resume's content aligns with a job description. The scikit-learn library in Python provides efficient implementations for TF-IDF vectorization and cosine similarity calculation.

3.5. Streamlit for Web Application Development

Streamlit is an open-source Python library that simplifies the process of creating interactive web applications for machine learning and data science projects. Its "Python-only" approach means developers can build sophisticated UIs without needing extensive knowledge of HTML, CSS, or JavaScript. Streamlit's rapid prototyping capabilities and straightforward component model make it an ideal choice for quickly deploying and showcasing AI models and data analysis tools. It handles the web server, frontend rendering, and interactivity, allowing the developer to focus on the core logic.

4. System Design and Architecture

The AI Resume Analyzer & Job Match Platform follows a modular and component-based architecture, designed for clarity, maintainability, and local execution.

4.1. High-Level Architecture

The system can be visualized as having three main layers:

1. **User Interface (Frontend):** Built with Streamlit, this layer handles user interaction, including file uploads and text input, and displays the analysis results.
2. **Application Logic (Backend - Python):** This is the core Python script (app.py) that orchestrates the entire process. It receives inputs from the UI, calls various helper functions for text processing and similarity, and interacts with the local AI model.
3. **AI/NLP Services (Local Backend):** This layer consists of the spaCy NLP library (loaded directly into the Python application) and the Ollama server (running as a separate background process), which hosts the Large Language Model (Gemma:2b)

4.2. Component Breakdown

- **app.py (Main Application Script):**
  - **Streamlit UI Components:** st.file_uploader, st.text_area, st.button, st.title, st.header, st.markdown, st.spinner, st.success, st.info, st.warning, st.error.
  - **File Handling:** Manages temporary storage of uploaded resume files.
  - **Text Extraction Functions:** extract_text_from_pdf, extract_text_from_docx.
  - **Text Preprocessing Function:** preprocess_text_spacy (utilizes spaCy).
  - **Similarity Calculation Function:** calculate_cosine_similarity (utilizes scikit-learn).
  - **AI Interaction Function:** analyze_with_ollama (utilizes ollama Python client).
  - **Orchestration Logic:** Controls the flow from input to processing to output.
- **venv/ (Virtual Environment):** An isolated Python environment containing all project-specific dependencies (streamlit, spacy, pypdf, python-docx, ollama, scikit-learn, numpy).
- **temp_uploads/ (Temporary Directory):** A local folder used to temporarily store uploaded resume files before text extraction. This folder is ignored by Git.
- **requirements.txt:** Lists all Python package dependencies with their exact versions, ensuring reproducibility of the environment.
- **.gitignore:** Specifies files and directories that Git should ignore (e.g., venv/, temp_uploads/, __pycache__/).
- **README.md:** Comprehensive documentation for the project, including setup, usage, features, and contribution guidelines.

4.3. Data Flow

1. **User Input:** The user uploads a resume file (PDF/DOCX) and pastes a job description into the Streamlit web interface.
2. **File Storage:** The uploaded resume file is temporarily saved to the temp_uploads/ directory on the local machine where the Streamlit app is running.
3. **Text Extraction:** Based on the file extension, either pypdf or python-docx extracts the raw text content from the temporary resume file.
4. **Temporary File Deletion:** The temporary resume file is immediately deleted from temp_uploads/ after text extraction for privacy and cleanliness.
5. **Text Preprocessing (for Similarity):** Both the extracted resume text and the job description text are sent to the preprocess_text_spacy function. This function uses spaCy to clean, tokenize, lemmatize, and remove stopwords from the text, preparing it for numerical analysis.
6. **Cosine Similarity:** The preprocessed texts are then fed into the calculate_cosine_similarity function, which uses TfidfVectorizer to convert them into numerical vectors and then computes their cosine similarity. The result is converted into a percentage match score.
7. **AI Prompt Construction:** The raw (or lightly processed) resume text and job description are combined into a detailed prompt string, specifically designed to guide the LLM's response.
8. **Ollama API Call:** This prompt is sent via the ollama Python client library to the Ollama server running locally.
9. **LLM Inference:** The Ollama server, using the loaded gemma:2b model, processes the prompt and generates a natural language response containing the AI-powered feedback.
10. **Display Results:** The match percentage and the AI-generated feedback are displayed back to the user on the Streamlit web interface.

5. Implementation Details

This section elaborates on the technical implementation of key components within the app.py script and the overall development setup.

5.1. Development Environment Setup

The project is developed using **Visual Studio Code (VS Code)** as the primary Integrated Development Environment (IDE). VS Code offers excellent Python support, including syntax highlighting, intelligent code completion (IntelliSense), debugging capabilities, and an integrated terminal.

Python Version: Python 3.11.7 is used, providing a balance of modern features, performance improvements, and long-term support.

Virtual Environments: A dedicated Python venv (virtual environment) is created for the project. This is a crucial best practice that isolates project dependencies, preventing conflicts with other Python projects or system-wide Python installations. All required libraries are installed within this isolated environment.

Ollama Installation: The Ollama server is installed directly on the user's operating system (macOS, Windows, or Linux). This server runs as a background process and exposes an API (typically on http://localhost:11434) that the Python application interacts with. The gemma:2b LLM model is downloaded and managed by Ollama.

5.2. Document Text Extraction

The platform supports two common resume formats: PDF and DOCX.

- **PDF Extraction:** The pypdf library is used. When a PDF file is uploaded via Streamlit's st.file_uploader, its content is read into a PdfReader object. The extract_text() method is then iterated over each page to concatenate the full text. Error handling is included to catch issues with corrupted or image-only PDFs.
- **DOCX Extraction:** The python-docx library handles DOCX files. Similar to PDFs, the uploaded file is read into a Document object, and text is extracted by iterating through its paragraphs.

A temporary directory (temp_uploads) is used to save the uploaded files to disk before processing, as some parsing libraries require a file path. These temporary files are promptly deleted after text extraction to maintain privacy and system cleanliness.

5.3. Text Preprocessing with spaCy

For accurate text matching, raw text needs to be cleaned and normalized. The spaCy library is employed for this purpose.

- **Loading Model:** The en_core_web_sm (small English web model) is loaded once at the application's start. This model provides linguistic annotations.
- **Lowercasing:** All text is converted to lowercase to ensure that "Python" and "python" are treated as the same word.
- **Tokenization:** Text is broken down into individual words or tokens.
- **Lemmatization:** Words are reduced to their base or dictionary form (e.g., "running," "ran," "runs" all become "run"). This helps in matching conceptually similar terms.
- Stopword and Punctuation Removal: Common words (like "the," "is," "and") and punctuation are removed as they typically don't carry significant meaning for matching purposes.
  The preprocess_text_spacy function encapsulates these steps, returning a clean, space-separated string of relevant tokens.

5.4. Cosine Similarity Calculation

The core quantitative matching is performed using TF-IDF and Cosine Similarity from scikit-learn.

- **TF-IDF Vectorization:** TfidfVectorizer is initialized. It learns the vocabulary from both the preprocessed resume and job description. It then transforms these texts into numerical vectors where each dimension corresponds to a word, and the value represents the word's importance (Term Frequency-Inverse Document Frequency). Words that are frequent in one document but rare across both are given higher weights.
- **Cosine Similarity:** The cosine_similarity function is applied to the two TF-IDF vectors. This function measures the cosine of the angle between the two vectors. A higher cosine value (closer to 1) indicates greater similarity between the documents. The result is then converted to a percentage for user-friendly display. This method effectively captures the thematic relevance between the resume and the job description.

5.5. AI-Powered Feedback Generation with Ollama

This is where the "intelligent" feedback comes from.

- **Prompt Engineering:** A detailed prompt is constructed. This prompt is critical as it instructs the LLM (Gemma:2b) on its role (expert career advisor), the task (analyze resume vs. job description), the desired output format (bullet points, bolding), and **crucially, specific instructions to avoid generic advice** (e.g., "DO NOT suggest adding sections if they are clearly discernable," "PRIORITIZE specific advice"). The raw resume and job description text (limited to 4000 characters each to fit within typical LLM context windows) are embedded directly into this prompt.
- **Ollama API Call:** The ollama.chat() function is used to send this prompt to the locally running Ollama server. The model parameter specifies gemma:2b. The stream=False argument ensures the function waits for the complete response from the LLM.
- **Response Handling:** The LLM's generated text is extracted from the response object and displayed directly in the Streamlit UI using st.markdown, which interprets any Markdown formatting in the LLM's output. Error handling is included to inform the user if Ollama is not running or the model is unavailable.

5.6. Streamlit User Interface

Streamlit provides the interactive frontend.

- **Page Configuration:** st.set_page_config is used to set the page title and layout (wide for better use of screen space).
- **Headers and Markdown:** st.title, st.header, and st.markdown are used to structure the page content, provide instructions, and display results with rich text formatting.
- **Input Widgets:**
    - st.file_uploader: Allows users to upload PDF or DOCX resume files.
    - st.text_area: Provides a multi-line input box for pasting the job description.
    - st.button: Triggers the analysis process.
- **Output Widgets:**
    - st.spinner: Provides visual feedback to the user while the application is processing, indicating that the AI is working.
    - st.success, st.info, st.warning, st.error: Used to display the match score with appropriate visual cues (emojis, colors) based on the percentage.
    - st.markdown: Displays the detailed AI-generated feedback, preserving any Markdown formatting from the LLM's response.
- **Conditional Logic:** The main processing block is wrapped in an if analyze_button: statement, ensuring that analysis only occurs when the button is clicked and inputs are present.

6. Testing and Results

Testing involved both functional verification and qualitative assessment of the AI's output.

6.1. Testing Methodology

- **Unit Testing (Implicit):** Individual functions (e.g., extract_text_from_pdf, calculate_cosine_similarity) were implicitly tested during development by running small code snippets.
- **Integration Testing:** The primary testing involved running the complete Streamlit application (python -m streamlit run app.py) and interacting with the UI.
- **Sample Data:** A variety of dummy and real-world (anonymized) resumes and job descriptions were used. This included:
    - Resumes in both PDF and DOCX formats.
    - Job descriptions with varying levels of detail and keyword density.
    - Resumes with clear matches, partial matches, and very low matches to job descriptions.
- **Performance Testing (Observational):** The responsiveness of the UI and the processing time for AI feedback were observed.
- **Error Handling:** Testing involved scenarios like uploading unsupported file types, empty inputs, or the Ollama server not running, to ensure appropriate error messages are displayed.

6.2. Sample Interactions and Outputs

A typical interaction involves:

1. **User Uploads Resume & Pastes Job Description:**
    - **Input (Resume):** A DOCX file detailing a "Software Engineer" with Python, Flask, SQL skills.
    - **Input (Job Description):** Text for a "Junior Python Web Developer" role requiring Flask, SQL, and Git.
2. **User Clicks "Analyze & Match My Resume!":**
    - st.spinner activates.
3. **Match Score Output:**
    - **Example Output:** 👍 **Good Match! Your resume matches 68.5% of the job description.** You're on the right track! A few tweaks could make it even better.
    - This score provides a quick, quantitative assessment.
4. **AI-Powered Tailoring Suggestions (from Ollama):**
    - st.spinner activates again for the LLM call.
    - **Example Output (after prompt refinement):**
      Strengths (What Matches Well):
      * **Python Proficiency:** Your resume clearly highlights strong Python skills, which is a core requirement.
      * **Web Development with Flask:** Explicit mention of Flask aligns perfectly with the "Python

Web Developer" aspect.
* **Database Management (SQL):** Your experience with SQL directly addresses the job's requirement for database skills.

Areas for Improvement (What's Missing or Weakly Highlighted):
* **Version Control (Git):** While common, Git isn't explicitly mentioned. Adding "Proficient in Git for version control" would strengthen this area.
* **Testing Frameworks:** The job might imply a need for unit/integration testing experience (e.g., Pytest, unittest), which isn't detailed.
* **Deployment Experience:** If the job involves deploying web apps, mentioning experience with cloud platforms (AWS, Azure, GCP) or Docker would be beneficial.

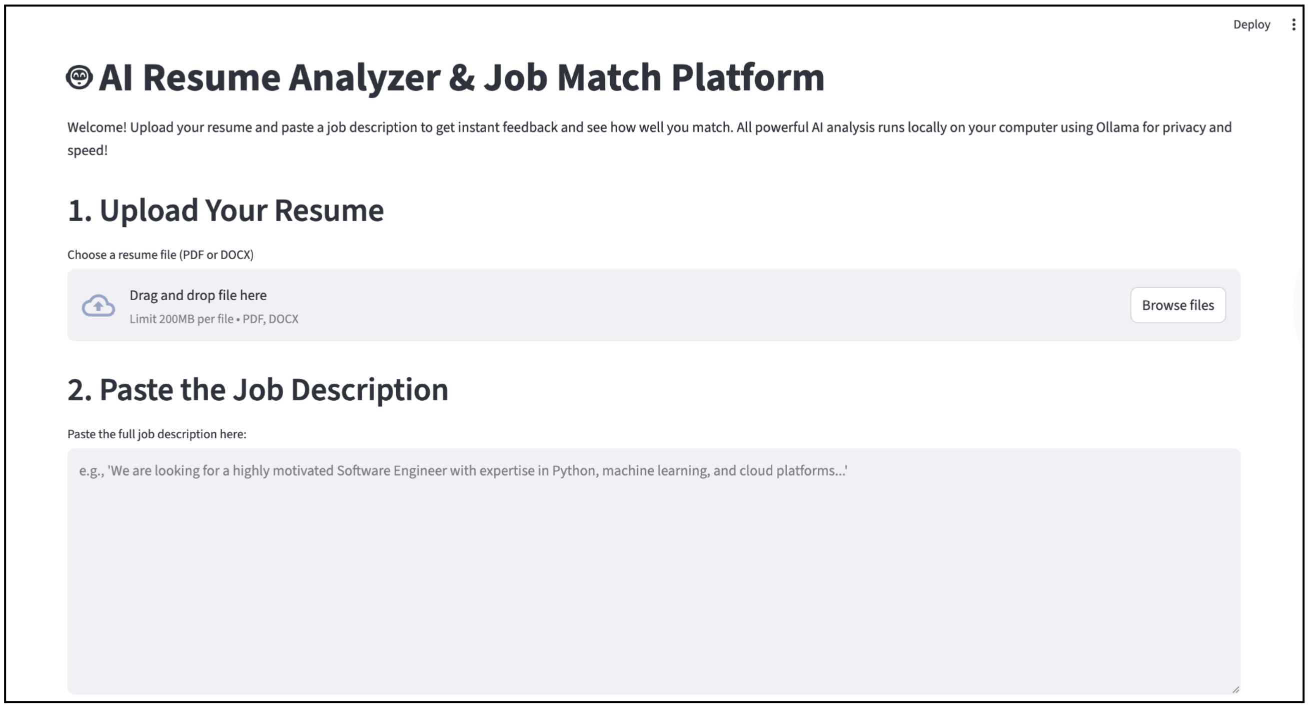
Tailoring Suggestions (How to Enhance Existing Content):
* **Quantify Achievements:** For your web application and data analysis projects, revise bullet points to include metrics. E.g., instead of "Developed web applications," try "Developed and deployed Flask web applications that improved user engagement by 15%."
* **Expand Project Details:** For each project, add 1-2 more bullet points detailing your specific contributions and the technologies used, directly linking them to job requirements.
* **Skills Section Enhancement:** If your skills are embedded, consider a dedicated "Technical Skills" section at the top, listing Python, Flask, SQL, and adding Git, Pytest, Docker if applicable.
Overall Fit (Summary):
Your resume shows a strong foundation for a Junior Python Web Developer. Focusing on quantifying your project impacts, explicitly mentioning version control, and potentially adding details on testing/deployment will significantly enhance your candidacy.

- This output demonstrates the AI's ability to provide contextually relevant and actionable advice, guided by the refined prompt.

6.3. Performance Observations

- **Initial Setup (Ollama Model Download):** The ollama run gemma:2b command involves downloading a ~1.5 GB model, which is a one-time process and can take several minutes depending on internet speed.
- **Application Startup:** Streamlit applications start relatively quickly (a few seconds).
- **Text Extraction:** For typical resumes (1-3 pages), text extraction is almost instantaneous.
- **Cosine Similarity:** This calculation is very fast, taking milliseconds.
- **Ollama Inference:** The most time-consuming part is the LLM inference. For the gemma:2b model running locally, responses typically take **5-30 seconds**, depending on the complexity of the prompt, the length of the input texts, and the user's CPU/GPU hardware. The st.spinner effectively manages user expectations during this period.
- **Watchdog Module:** As suggested by Streamlit, installing watchdog can improve live reloading performance during development.

7. Screenshots

■ 

AI Resume Analyzer & Job Match Platform

Welcome! Upload your resume and paste a job description to get instant feedback and see how well you match. All powerful AI analysis runs locally on your computer using Ollama for privacy and speed!

1. Upload Your Resume

Choose a resume file (PDF or DOCX)

Drag and drop file here
Limit 200MB per file • PDF, DOCX

Browse files

2. Paste the Job Description

Paste the full job description here:

e.g., 'We are looking for a highly motivated Software Engineer with expertise in Python, machine learning, and cloud platforms...'

- **Purpose:** To showcase the core intelligent output of the platform, demonstrating the value of the LLM integration.

8.1. Python Environment and Dependency Management

- **Challenge:** A common issue was ModuleNotFoundError: No module named 'streamlit.cli' when running streamlit run app.py, especially on systems with Anaconda installed. This occurred because the system's PATH was prioritizing an older or incompatible streamlit executable from Anaconda's base environment over the one installed in the project's virtual environment.
- **Solution:** The most effective solution was to explicitly invoke Streamlit via Python's module runner: python -m streamlit run app.py. This ensures that the streamlit module *within the currently active virtual environment* is used, bypassing any PATH conflicts. Additionally, meticulous verification of active virtual environments (which python, pip show streamlit) and aggressive reinstallation (pip install --upgrade --force-reinstall streamlit) were used to ensure the correct library versions were in place.

8.2. Generic AI Feedback from LLMs

- **Challenge:** Initially, the LLM (Gemma:2b) tended to provide generic resume advice (e.g., "Consider adding a 'Skills' section" or "Quantify your accomplishments") even when such elements were already present in the resume. This is a common characteristic of general-purpose LLMs, which prioritize plausible generation over precise factual verification within a limited context window.
- **Solution:** This was primarily addressed through **advanced prompt engineering**. The prompt sent to Ollama was significantly refined to include "CRITICAL INSTRUCTIONS FOR ACCURACY." These instructions explicitly directed the LLM to:
  - Avoid suggesting additions if elements were discernable.
  - Focus on *improving* or *aligning* existing content.
  - Avoid advice on elements not present in the input (like cover letters).
  - Prioritize specific advice over generic tips.
  - Frame quantification suggestions as revisions to existing points.
    While not a perfect solution (LLMs can still "hallucinate" or misinterpret), this iterative prompt refinement significantly improved the relevance and specificity of the AI-generated feedback. For ultimate precision, fine-tuning a model on domain-specific data would be the next step, but it's outside the current project scope.

9. Future Work and Enhancements

The current platform provides a strong foundation, but several enhancements can further improve its functionality, accuracy, and user experience:

- **Advanced Resume Parsing:** Implement more sophisticated NLP techniques (e.g., Named Entity Recognition, custom entity extraction) to reliably identify and extract specific sections (e.g., "Education," "Work Experience," "Skills," "Projects") and fields (e.g., dates, company names, job titles) from resumes. This structured data could then be used for more precise matching and targeted AI analysis.
- **Skill Gap Analysis with Taxonomy:** Integrate a predefined skill taxonomy or ontology. This would allow the system to not just match keywords but understand related skills (e.g., "Python" implies "programming," "data analysis") and identify broader skill gaps or overlaps.
- **User Authentication and Profile Management:** Implement a user login system to allow users to save multiple resumes and job descriptions, track their application history, and view past analyses. This would require integrating a database (e.g., Firestore, SQLite).
- **Enhanced UI/UX:** While Streamlit is great for rapid development, exploring custom Streamlit components or integrating with a more flexible frontend framework (like React.js with a Python backend using Flask/FastAPI) could allow for more sophisticated and visually rich user interfaces.
- **Real-time Job Board Integration:** Develop modules to scrape job postings from popular job boards, allowing users to analyze jobs directly from the platform without manual copy-pasting.
- **Interview Preparation:** Extend the LLM's capabilities to generate mock interview questions based on the resume and job description, or even provide feedback on recorded responses.
- **Performance Optimization:** For larger LLMs or higher concurrency, explore optimizing Ollama's

setup (e.g., GPU utilization, quantization) or consider hybrid approaches that offload some processing to cloud services if privacy concerns are mitigated.

- **Multi-Language Support:** Extend NLP models and LLM prompts to support resume analysis and job matching in multiple languages.

10. Conclusion

The "AI Resume Analyzer & Job Match Platform" successfully demonstrates the power of combining open-source NLP tools (spaCy), a locally hosted Large Language Model (Ollama with Gemma:2b), and a rapid web development framework (Streamlit) to create an intelligent and privacy-conscious solution for resume optimization and job matching. The platform provides valuable quantitative (cosine similarity) and qualitative (AI-generated feedback) insights, directly addressing the pain points of job seekers and recruiters.

Through careful prompt engineering and robust error handling, the system delivers actionable advice, proving the viability of local AI for sensitive applications. This project serves as a strong foundation for future development, with clear pathways to enhance its intelligence, user experience, and integration capabilities, ultimately contributing to a more efficient and effective job market.

11. References and Acknowledgments

- **Python:** https://www.python.org/
- **Visual Studio Code:** https://code.visualstudio.com/
- **Streamlit:** https://streamlit.io/
- **Ollama:** https://ollama.com/
- **spaCy:** https://spacy.io/
- **pypdf:** https://pypdf.readthedocs.io/
- **python-docx:** https://python-docx.readthedocs.io/
- **scikit-learn:** https://scikit-learn.org/
- **Git:** https://git-scm.com/
- **GitHub:** https://github.com/