AI-Powered Resume and Job Matching System

Students:

- RODOLPHE BIELEU
- SASCHA CAUCHON

Class: DIA2

Abstract

The smart resume and job matcher is an AI system designed to automate and enhance the recruitment process. Organizations receive hundreds of applications for every open position and HR face an overwhelming task of reviewing each resume against job requirements. This project addresses that challenge by combining modern natural language processing techniques with traditional matching algorithms to create a system that can intelligently pair candidates with suitable job opportunities.

This report provides a comprehensive overview of the system, explaining how it was built, why certain technical decisions were made, and how the various components work together to deliver meaningful results.

1. Introduction and context

1.1 The problem

The hiring process has long been a problem in organizational growth. When a company posts a job opening especially at big companies with popular positions the number of applications can be very high. A single job posting might attract anywhere from 100 to over 1,000 applicants. HR simply cannot give each application the attention it deserves within reasonable timeframes. If we look on the candidates side, they have little information about if they are a good fit for that position.

Traditional keyword-based filtering systems often miss qualified candidates whose resumes use different terminology to describe similar skills. For instance, a candidate might describe their work with "neural networks" while the job posting asks for "deep learning" experience. These terms are semantically equivalent but a simple keyword matcher would fail to connect them. Another problem is that keyword systems might rank irrelevant candidates highly simply because they stuffed their resumes with popular buzzwords.

This project was created to solve these problems by building a system that truly understands the meaning behind text not just the literal words. We use semantic understanding through embeddings and LLMs. The system can make nuanced judgments about candidate-job fit that more closely mirror how a skilled human recruiter would evaluate applications.

1.2 Project goals

First, we wanted to create a system that could parse resumes and job descriptions automatically. We wanted to extract all relevant information regardless of how the documents were formatted. Second, we aimed to build a matching engine that goes beyond simple keyword comparison to understand the deeper meaning and context of both candidate qualifications and job requirements. Third, we wanted to provide explainable results because matching scores alone are not particularly useful without understanding why a candidate was deemed a good or poor fit. Finally, we wanted the system to be flexible and accessible allowing users to choose from different AI providers based on their needs and constraints.

2. System architecture

2.1 Overview of components

The system follows an architecture where documents flow through several processing stages before producing final match results. This design allowed each component to be developed, tested, and improved independently. If we discover a better way to parse resumes in the future, we can swap out that component without affecting the rest of the system.

At the highest level the system consists of four main layers. The first layer handles document ingestion and parsing with files in different formats and different content. The second layer manages embedding generation where text is transformed into numerical vectors that capture semantic meaning. The third layer performs matching and analysis comparing candidates against jobs using multiple dimensions. The fourth and final layer generates human-readable explanations using large language models.

2.2 Document parsers

The document parsing layer is the foundation of the entire system. This is important because without accurate extraction of information we would not produce meaningful results.

The resume parser was designed to extract over twenty different fields from candidate documents. These include obvious elements like name, email, and phone number, but extend to more relevant information such as individual project descriptions, certifications, and even personal interests. The parser recognizes that resumes come in many formats and layouts, so it uses multiple strategies to find information. It looks for common section headers like "Experience" or "Education" to locate relevant content, but also employs pattern matching to identify things like email addresses and phone numbers regardless of where they appear.

The job description parser performs a similar extraction process but focuses on the elements that matter for job postings. This includes the job title and company name but also required and preferred skills, experience level requirements, salary information, and the technology stack used by the team. One important distinction the parser makes is between hard requirements and nice-to-have qualifications. This distinction matters significantly during matching because missing a required skill is far more serious than lacking a preferred one.

Both parsers support multiple file formats including PDF, DOCX, and txt files. For PDF parsing, the system uses pdfplumber as the primary tool with PyPDF2 as a fallback, ensuring reliable text extraction from even complex PDF layouts.

2.3 Embedding service

Embeddings are numerical representations of text that capture meaning rather than just literal content. The system supports 2 embedding providers. SentenceTransformers is the default choice because it runs entirely locally without requiring any internet connection or API costs. The default model (all-MiniLM-L6-v2) produces 384-dimensional vectors and offers an excellent balance between quality and speed. Finally, Google's embedding API provides cloud-based embeddings for users who want the convenience of a hosted solution and are willing to use an API key.

The embedding service follows the provider pattern in its design, meaning all 2 providers implement the same interface. This makes switching between providers as simple as changing a configuration setting. The service normalizes all vectors using L2 normalization before

returning them, which ensures that cosine similarity calculations work correctly regardless of which provider generated the embeddings.

2.4 Matching and analysis engine

The matching engine combines semantic similarity with structured analysis to produce comprehensive match scores. Early versions of the system relied solely on embedding similarity but when we tested we saw that this approach missed important factors that experienced recruiters always consider.

The current system uses a weighted multi-dimensional scoring approach. Semantic similarity from embeddings accounts for thirty percent of the final score. Skills matching contributes twenty-five percent and examines whether the candidate possesses the specific skills listed in the job requirements. Experience analysis provides twenty percent and considers both the total years of experience and the relevance of past roles. Education matching adds ten percent by comparing degree levels and fields of study. Certifications contribute eight percent by checking whether required certifications are present. Finally, project relevance adds seven percent by examining whether the candidate's past projects used technologies relevant to the position.

These weights were determined through experimentation. The system also categorizes matches into five fit levels ranging from Excellent for scores above eighty percent down to Poor for scores below thirty-five percent. These labels provide quick insights without requiring users to interpret numerical scores.

2.5 Explanation generation

Match scores alone tell users that a candidate is a good or poor fit but they do not explain why. The explanation chain addresses this limitation by using LLMs to generate detailed and human-readable analyses of each match.

The explanation system uses LangChain as its orchestration framework. The system supports two LLM backends: Ollama for local inference and Google's Gemini for cloud-based processing.

The prompt template used for explanation generation was carefully designed to produce honest and actionable feedback. It instructs the model to first check hard requirements like specific certifications or degree levels, then analyze skill gaps, evaluate experience relevance, and finally provide a clear recommendation.

2.6 UI

The primary interface is a Streamlit web application that offers an intuitive graphical experience. Users can upload multiple resumes and job descriptions, trigger parsing and matching operations with button clicks, and view results with rich formatting including color-coded scores and collapsible detail sections. A Jupyter notebook is also provided that walks through the complete workflow step by step.

3. Design choices and rationale

3.1 Why embeddings

Keyword systems work by checking whether specific words appear in documents. While fast and interpretable, they suffer from two major problems that significantly impact recruitment scenarios.

The first problem is vocabulary mismatch. Different people use different words to describe the same concepts. A candidate might write about their experience with "convolutional neural networks" while a job posting asks for "CNN expertise." A keyword system would miss this match entirely. The second problem is context insensitivity. Keywords cannot distinguish between different uses of the same word. The word "Python" might refer to the programming language or to a snake, depending on context.

Embeddings solve both problems. Texts with similar meanings end up close together in this space regardless of the exact words used. This allows the system to recognize that "machine learning" and "ML" refer to the same thing, or that "five years of Python development" and "extensive Python experience" convey similar qualifications.

3.2 Multi-dimensional scoring

Relying solely on semantic similarity would be insufficient for practical recruitment scenarios. While embeddings capture overall document similarity, they do not explicitly account for structured requirements that recruiters care deeply about.

Consider a job that requires a PhD and five years of experience. A candidate with a Master's degree and three years of experience might have a resume that is semantically very similar to the job description, producing a high embedding similarity score. However, they fundamentally do not meet the posted requirements. That’s why we assigned weights to each component based on what the job wants.

3.3 Provider flexibility

A key design decision was supporting multiple providers for both embeddings and language models rather than hardcoding a single solution. This flexibility serves several important purposes. It respects user constraints. Some organizations don’t feel comfortable sharing data to external services and that’s why there are local solutions. Others want convenience and quality which makes models from cloud APIs more attractive. By using both of these we accommodate different requirements. Also if one model is down then a user can easily switch to another one. And it's also good when you want to compare results through different providers and see which one you like best.

3.4 Dataclass-based structures

Throughout the codebase, parsed documents and match results are represented using Python dataclasses rather than plain dictionaries. Dataclasses provide type hints that enable better IDE support including autocomplete and error detection. They make the expected structure of data explicit, serving as documentation that is always synchronized with the actual code. They also support default values and automatic generation of methods like comparison and string representation.

4. Data processing pipeline

4.1 Text extraction

The first step in processing any document is extracting raw text from its file format. This seemingly simple task is actually quite complex due to the variety of formats and layouts encountered in real-world documents.

For PDF files, the system uses pdfplumber as the primary extraction tool. When pdfplumber fails, the system falls back to PyPDF2, which uses different extraction techniques that sometimes succeed where pdfplumber struggles. For Word documents, the python-docx library reads the underlying XML structure and reconstructs the text content. Plain text files are read directly with automatic encoding detection to handle various character sets.

4.2 Structured information extraction

Once raw text is available, the parsers extract structured information using a combination of pattern matching and section analysis. The general approach involves identifying section boundaries using header keywords, then applying specialized extraction logic within each section.

For contact information, regular expressions identify patterns like email addresses, phone numbers, and URLs matching common professional profiles like LinkedIn and GitHub. Skills extraction uses a dual approach. First, the parser looks for explicit skills sections and extracts listed items. Second, it scans the entire document for mentions of known skills from a predefined vocabulary of several hundred common technical and soft skills. This hybrid approach catches skills whether they appear in a dedicated section or are mentioned in context within experience descriptions. Experience extraction looks for patterns indicating work history entries, such as job titles followed by company names and date ranges. Education extraction follows similar patterns, looking for degree names, institution names, graduation years, and GPAs.

4.3 Embedding generation

After structured information is extracted, the parser generates a text representation optimized for embedding. This representation concatenates all relevant sections with appropriate labels, creating a document that captures the candidate's full profile in a form suitable for semantic

analysis. For local providers, this happens entirely on the user's machine using pre-downloaded models. For cloud providers, the text is sent via API and the resulting vector is returned. All providers normalize their output vectors, ensuring consistency in downstream similarity calculations.

4.4 Matching process

The system calculates a final match score by comparing resumes and job descriptions across several dimensions using a weighted formula. It uses cosine similarity between embedding vectors for broad semantic matching while performing specific checks on structured data.

For skills, it measures the overlap percentage and identifies gaps. Experience is evaluated by total years, industry overlap, and title relevance. Education is ranked on an ordinal scale (PhD to Bachelor's) while checking for field alignment. To handle naming variations in certifications, the system uses partial string matching to ensure different aliases for the same credential are recognized. Finally, candidates are sorted by their scores and assigned fit levels based on defined thresholds.

5. LLM provider selection

5.1 The role of LLMs

LLMs serve two distinct purposes in this system. First, some providers offer embedding capabilities that compete with dedicated embedding models. Second, LLMs generate the detailed explanations that make match results actionable.

For embeddings, the project primarily relies on SentenceTransformers because these models were specifically trained for the task of producing meaningful sentence embeddings. For explanation generation, the expressive capabilities of modern LLMs truly shine. The system needs to analyze structured match data and produce coherent, helpful prose that highlights strengths, identifies gaps, and provides actionable recommendations.

5.2 Google Gemini for cloud inference

For users who prefer cloud services, the system integrates with Google's Gemini models through the Google Generative AI API. Gemini offers excellent quality with fast response times and requires only an API key for access.

The default model is Gemini 2.5 Flash, which balances quality and cost effectively for the explanation generation task. Users can specify alternative models if desired, such as the more capable Gemini Pro. Cloud inference eliminates hardware requirements and provides consistent performance regardless of the user's local machine capabilities. The main considerations are cost and sending data over to other services.

6. Evaluation and results

6.1 Testing methodology

The system was evaluated using a collection of sample resumes and job descriptions designed to test various matching scenarios. The samples include candidates with different levels of experience, from entry-level to senior positions. Job descriptions span multiple domains including data science, software engineering, and business analysis.

Key test scenarios included exact matches where candidates have all required qualifications, partial matches where candidates have most but not all requirements, mismatches where candidates lack critical qualifications, and cross-domain scenarios where candidates are applying for positions somewhat outside their primary field.

6.2 Explanation quality

The explanations generated by the LLM component received positive feedback during evaluation. They successfully identify key strengths by referencing specific achievements and experiences from the resume. They clearly articulate gaps by naming missing skills, insufficient experience, or education mismatches. They provide actionable recommendations that help both recruiters and candidates understand potential next steps.The prompts were iterated several times to achieve honest and balanced outputs.

7. Future improvements

7.1 Enhanced document understanding

The current parsers use pattern matching and heuristics to extract information from documents. While effective for typical resume and job description formats, they can struggle with highly unusual layouts or unconventional organization. A natural enhancement would be incorporating vision-language models or layout-aware document understanding systems. These could analyze the visual structure of PDF documents rather than just extracted text.

7.2 Fine-tuned embedding models

The system currently uses general-purpose embedding models trained on broad text. While these work well, domain-specific models trained specifically on recruitment documents could provide better performance. This would require access to historical hiring data, which presents both practical and privacy challenges, but could significantly improve matching accuracy.

7.3 Interactive feedback loop

Currently, the system produces results but does not learn from user feedback. Adding a feedback mechanism where recruiters indicate whether suggested matches were actually helpful would enable continuous improvement.

7.4 Integration Capabilities

Real-world deployment would benefit from integrations with applicant tracking systems, job boards, and HR information systems. API endpoints could enable organizations to incorporate the matching engine into their existing workflows rather than using it as a standalone tool. Also we could use docker for deployment and even build a better interface with a web programming framework such as Next.js.

7.5 Bias detection and mitigation

Like all AI systems, there is potential for the matcher to exhibit biases present in training data or embedding models. Future work should include systematic evaluation for bias across protected categories and development of mitigation strategies where bias is detected.

Appendix: Technology Stack

The following technologies form the foundation of the system:

- **Core Framework**: Python 3.10+, providing the programming foundation with modern type hints and dataclass support.
- **Document Parsing**: pdfplumber and PyPDF2 for PDF files, python-docx for Word documents, with automatic encoding detection for text files.
- **Embeddings**: SentenceTransformers library with the all-MiniLM-L6-v2 model as default, Ollama client library for local embedding models, and Google Generative AI SDK for cloud embeddings.
- **Large Language Models**: LangChain framework for orchestration, langchain-ollama for local inference, and langchain-google-genai for cloud inference.
- **Numerical Computing**: NumPy for vector operations, scikit-learn for additional ML utilities.
- **Data Validation**: Pydantic for structured data validation throughout the pipeline.
- **User Interface**: Streamlit for the web application, Jupyter for the interactive notebook demo.
- **Configuration**: python-dotenv for environment variable management, enabling flexible configuration across environments.