
Project Overview

This system is a modular platform for evaluating how well resumes fit job descriptions (JDs) using deterministic, semantic, and AI feedback. The backend (Flask) processes uploads, text extraction, skill parsing and scoring, while the frontend (Streamlit) offers an interactive dashboard for users. Semantic search, embedding storage, and advanced parsing ensure high accuracy and actionable feedback.embeddings.py+7

File & Folder Reference Table

Filename	Description
app.py	Main Flask backend API, handling upload, parsing, evaluation requests.
streamlit_app.py	Streamlit dashboard—file upload, evaluation, and results display, connects to Flask API.
parser.py	NLP utility: splits resumes/JDs into sections; extracts and normalizes skills data.
scoring.py	Implements matching and scoring logic: skill overlap, semantic similarity, verdicts.
suggestion.py	Provides AI-generated advice and summary suggestions for improving fit.
seed_skills.py	Initializes a core skills list and exports as JSON for reference by parsers/functions.
requirements.txt	Dependency pinning—NLP, web, embedding, LLM, and database packages.
embeddings.py	Handles semantic embedding model loading, FAISS index storage, querying, normalization.
extractor.py	Unified DOCX, PDF, TXT text extraction—modular and temp-file safe.
evaluator.py	Orchestrates evaluation: runs extraction, parsing, scoring, verdicts, and missing skills.
README.md	(Short) General project info or usage introduction.

Module-by-Module Breakdown

app.py

- **uploadjd**: API POST; handles JD upload, text extraction, and skill parsing. JSON response.app.py
- **uploadresume**: API POST; handles resume upload, links resume to JD, triggers evaluation, returns fit results.app.py
- **geteval**: API GET; retrieves evaluation details for given resume ID.app.py

streamlit_app.py

- Interactive scripts, no classes. Uses Streamlit widgets for:
    - JD upload (file, ID, must/nice skills).streamlit_app.py
    - Resume upload (file, ID, JD reference).streamlit_app.py
    - Collects/display evaluation data from backend and errors.streamlit_app.py

parser.py

- **splitbysections(text)**: Separates document into logical sections (skills, experience, etc.).parser.py
- **extractskillsfromtext(text, skilllist=None)**: Detects skills with regex, NLP, and fuzzy matching; optional list matching.parser.py
- **extracteducation(text)**: Finds education entries via search/regex.parser.py

scoring.py

- **hardmatchscore(jdskills, resumeskills)**: Calculates “must”/“nice” skill matches and weighted score.scoring.py
- **semanticscore(jdtext, resumetext, embedstore)**: Calculates semantic fit via embeddings.scoring.py
- **finalscore(hard,semantic,weights)**: Weighted sum of hard and semantic fit.scoring.py
- **verdictfromscore(score)**: Converts a score into qualitative verdict (“High”/“Medium”/“Low” fit).scoring.py

suggestion.py

- **generatefeedback(jdtext, resumetext, missingskills, topn=5)**: LLM-powered function generating:
    - Profile summary
    - Top 5 improvement actions

- Shortlisting suggestion for recruiters.suggestion.py

seed_skills.py

- **DEFAULT_SKILLS**: Tech skills list (static variable).seed_skills.py
- Writes skills to JSON for system-wide consistency.seed_skills.py

embeddings.py

- **Class EmbedStore**: Loads sentence-transformer model, persists FAISS index for fast semantic search/retrieval.embeddings.py
  - **embed(texts)**: Converts text(s) to embeddings.
  - **add(text, meta)**: Adds new entry to index with associated metadata.embeddings.py
  - **save()**: Persists updated FAISS index and metadata to disk for reuse.embeddings.py
  - **query(text, topk=5)**: Finds most similar entries in the embedding space.embeddings.py
- Parameters: customizable embedding dimensions, model type, index location, etc..embeddings.py

extractor.py

- **cleantext(text)**: Standardizes whitespace, removes headers/footers, normalizes input.extractor.py
- **extractpdf(filepathorbytes)**: Extracts and cleans text from PDFs, byte input or file path.extractor.py
- **extractdocx(filepathorbytes)**: Handles .docx extraction for both file and bytes input, with temp-file workaround.extractor.py
- **extracttextfromupload(uploadedfile)**: Smart dispatcher—uses file extension to call relevant sub-extractor.extractor.py

evaluator.py

- **evaluateresume(uploadedresume, jdtext, jdparsedskills)**: Pipeline function:
  - Extracts text, parses with section/skill methods, computes scores, computes verdict and missing skills.evaluator.py
  - Returns: final fit score, breakdown (hard vs semantic), verdict (“High”/“Medium”/“Low”), missing skills, per-section analysis.evaluator.py

---

System Architecture and Integration

- **File upload, parsing, and scoring** flow from the Streamlit UI, through Flask endpoints, then modularly through extraction (extractor.py), parsing (parser.py), scoring (scoring.py), embeddings (embeddings.py), and LLM-based feedback (suggestion.py).suggestion.py+7
- **Embeddings and semantic search** are persistent and updatable, supports large scale and tuning (with FAISS + SentenceTransformer backend).embeddings.py
- **All functions are testable** independently and use clear, minimal interfaces for ease of extension—new skill domains, new extractors, or different evaluation logic can be added with minimal friction.evaluator.py+5

Quickstart Usage Flow (for Hackathon Demo)

1. Launch backend API server (app.py).
2. Launch Streamlit dashboard (streamlit_app.py).
3. Upload a job description (JD) and parse skills (must/nice).streamlit_app.py
4. Upload candidate resumes—view scored match result, highlight missing skills, AI-powered improvement tips, and recruiter summary.suggestion.py+1
5. All major backend steps are visible and well-logged for both demo and debugging.
