Table of Contents

Chapter 1

Synopsis

The Resume Ranking System is an intelligent software solution developed using NLP techniques in Python. It aims to automate the shortlisting of resumes based on their relevance to a given job description. Using cosine similarity and keyword accuracy, this system evaluates PDF and DOCX resumes and ranks them accordingly. The application features a GUI built using Tkinter for user-friendly interaction, allowing easy job description input and resume folder selection. The results are displayed in a table format and can be exported as a CSV file.

Chapter 2

Introduction

In today's competitive job market, companies receive hundreds or even thousands of resumes for a single job opening. Manually reviewing and screening each resume against the job requirements is not only time-consuming but also introduces inconsistencies due to human bias, fatigue, and subjective judgment. To address these challenges, automation in the resume shortlisting process has become essential. A Resume Ranking System powered by Natural Language Processing (NLP) helps automate the screening process by evaluating how well a resume matches the given job description. Such systems significantly reduce the recruiter's workload, ensure consistency in candidate evaluation, and help prioritize the most relevant resumes. This project introduces a Resume Ranking System that uses cosine similarity and keyword accuracy techniques to compute a relevance score for each resume. The user interface is built using Python's Tkinter library, enabling recruiters to input a job description, select a folder of resumes (in PDF or DOCX format), and receive ranked results with export functionality.

2.1. Scope of the Document

This Low-Level Design (LLD) document describes the Internal Workings of the Resume Ranking System. It describes the data processing pipeline, embedding-based scoring method, GUI flow, validation strategies, and configuration options required for deployment. The document is intended for developers and testers who will implement and verify the system.

Specifically, the document includes:

- End-to-end analysis of the user interface and internal logic
- Component-level interaction graphs (sequence and flow).
- Design choices for each module (parser, vectorizer, GUI, etc.).
- Error management and fallback strategies
- UI validation and feedback techniques
- In-memory and optional permanent data management
- Testing Strategies and Future Improvement Considerations
- Justifications for using specific libraries and frameworks.

The scope includes not just the existing desktop implementation, but also plans for future migration to web-based platforms or ATS integration.

2.2 Intended Audience

This document is designed for a wide range of technical and semi-technical readers who contribute to or review the system design and development.

- **Software Developers:** Understand how to construct or extend the system using the modular architecture and component breakdown.
- **QA/Test Engineers:** Determine essential areas for testing and validation using the given logic and validations.
- **Technical mentors, project reviewers, and evaluators:** Assess the system's design

logic, maintainability, and correctness.

- **Future Contributors:** To quickly comprehend the system's aim and internal workings in order to improve or refactor it later.
- **Internship Assessors or Academic Evaluators:** Verify if low-level design documentation complies with academic and industry standards.

Readers are assumed to be familiar with Python, fundamental NLP approaches, and software development best practices.

2.3 System Overview

The Resume Ranking System is a similarity-based tool that uses natural language processing (NLP) to automatically screen resumes. It accepts unstructured input files (resumes and job descriptions), converts them to semantic vectors, and returns sorted results based on cosine similarity. It extracts features with pre-trained language models and matches candidates efficiently using supervised ranking logic.

Key Components:

- **Frontend GUI:** Built with Tkinter or Streamlit, HR personnel can upload files and check ranked results.
- **The Text Parser:** extracts text from uploaded resumes (PDF/DOCX) and job descriptions.
- **The Embedding Engine:** converts captured text into numerical vectors using SpaCy embeddings.
- **Similarity Calculator:** Calculates the cosine similarity scores between job descriptions and resume vectors.
- **Ranking Module:** Sorts resume according on similarity score.
- **Output Renderer:** Displays the ranked list in the interface, including names and scores.
- **The Validation Layer:** ensures that uploaded files are readable, non-empty, and in valid format.

Chapter 3
Background Research

Traditional hiring tools and Applicant Tracking Systems (ATS) often rely on simple keyword-based filtering to screen resumes. While this method is fast, it lacks the ability to understand the context in which those keywords appear. For example, the presence of the word “Python” in a resume doesn’t always mean the candidate is proficient in it — context matters.

Recent developments in Natural Language Processing have provided smarter solutions. Semantic similarity techniques — like cosine similarity using word embeddings — can identify not just keyword presence but also their meaning in context. NLP frameworks like **SpaCy** offer pre-trained language models that generate vector-based representations of text, allowing semantic comparison between job descriptions and resumes.

In our system, we combine:

- **Cosine Similarity**: To measure overall semantic relevance between the job description and resume text.
- **Keyword Matching Accuracy**: To ensure critical terms in the JD are present in the resume.
- **Boosting Mechanisms**: That give extra weight to resumes with higher keyword coverage.

By blending traditional and modern techniques, our approach improves both **precision** (correctly identifying relevant resumes) and **recall** (not missing potentially good candidates). The architecture is modular and extensible, allowing for future improvements like integration of BERT or OCR for scanned documents.

Chapter 4

Tools and technologies

• **Python 3.11**

• **Tkinter** - GUI development

Used for creating the graphical user interface (GUI) where users can input job descriptions, select folders, and view results.

• **Pandas**

Used for handling tabular data, creating and manipulating Data Frames, and exporting ranked results to CSV.

• **SpaCy** (en_core_web_md) - text vectorization

A powerful NLP library used to calculate cosine similarity and extract semantic information from job descriptions and resumes.

• **scikit-learn** - cosine similarity computation

• **PyMuPDF (fitz)**

Used for extracting text content from .pdf resumes in a reliable and efficient way.

• **docx2txt**

Used to extract clean text from .docx resumes.

• **OS & String Libraries**

For handling file paths, extensions, and text cleaning operations.

• **Stop words from SpaCy**

Used to filter out common words that are not meaningful for keyword matching (like "and", "with", etc.).

Chapter 5

Implementation

5.1 Application Design

The Resume Ranking System has been designed using a modular architecture that separates the user interface, text extraction, preprocessing, similarity calculation, ranking, and output rendering.

The main modules are:

- **Graphical User Interface (Tkinter):** Provides fields to enter job description, select resume folder, view results, and export CSV.
- **Text Extractor:** Extracts text from resumes in PDF (using PyMuPDF) and DOCX (using docx2txt) formats.
- **Preprocessing and Keyword Extraction:** Cleans input text, removes stopwords, and extracts relevant keywords from job description.
- **Similarity Engine:** Computes cosine similarity between job description and resumes using SpaCy embeddings.
- **Ranking Engine:** Combines similarity and keyword accuracy using a weighted formula to assign final scores.
- **Output Renderer:** Displays ranked resumes in GUI and enables CSV export.

5.2 Information Flow

The system follows a structured flow of information:

1. **Job Description Input:** The recruiter enters the job description into the GUI.
2. **Resume Collection:** The user selects a folder containing resumes in PDF/DOCX format.
3. **Text Extraction:** Each resume is processed and converted into plain text.
4. **Preprocessing:** Stopwords and irrelevant content are removed, and keywords are extracted from the job description.
5. **Similarity Computation:** SpaCy embeddings are used to compute cosine similarity between JD and resumes.
6. **Ranking:** Final scores are calculated using the weighted formula:

$$\textbf{Final Score} = 0.2 \times \textbf{Similarity} + 0.8 \times \textbf{Keyword Accuracy} + \textbf{Boost Final}$$

7. **Results:** The ranked resumes are displayed in the GUI and can be exported to CSV for offline use.

5.3 Components Design

1. GUI Module (main.py)

- Provides interface to input job description.
- Allows folder browsing for resumes.
- Displays ranked results in a Treeview table.
- Provides “Export to CSV” option.

2. Text Extractor Module (text_extractor.py)

- Extracts raw text from resumes.
- Uses **PyMuPDF** for PDF files.
- Uses **docx2txt** for DOCX files.

3. Preprocessing and Keyword Engine

- Tokenizes and cleans job description.
- Removes stopwords and special characters.
- Extracts relevant keywords for comparison.

4. Similarity and Ranking Engine (ranker.py)

- Converts text to embeddings using SpaCy.
- Computes cosine similarity between JD and resume.
- Calculates keyword accuracy (percentage of JD keywords found in resume).
- Applies weighted formula to generate final score.

5. Output Renderer

- Displays ranked resumes in GUI.
- Provides CSV export for offline usage.

5.4 Key Design Considerations

The following factors influenced the system design:

- **Technology Choice:**
    - Tkinter selected for GUI as it is lightweight and easy to integrate with Python applications.
    - SpaCy chosen for embeddings to capture semantic similarity efficiently.
    - PyMuPDF and docx2txt selected for reliable text extraction from resumes.

- **Ranking Strategy:**
    - A hybrid approach using both semantic similarity (cosine similarity) and keyword matching ensures balanced evaluation.
    - Higher weight (0.8) given to keyword accuracy to prioritize specific job requirements.
- **Performance:**
    - System processes ~100 resumes in under 30 seconds.
    - Optimized for efficient screening without compromising accuracy.
- **Scalability:**
    - Modular design allows future enhancements such as integration with BERT embeddings, OCR for scanned resumes, or ATS systems.

5.5 API Catalogue

Module	Function/Method	Description
text_extractor.py	**extract_text_pdf(path)**	Extracts text from PDF resumes using PyMuPDF
	extract_text_docx(path)	Extracts text from DOCX resumes using docx2txt
ranker.py	**compute_similarity(text1, text2)**	Calculates cosine similarity between JD and resume
	calculate_keyword_accuracy(text, keywords)	Computes keyword match accuracy
	rank_resumes(resumes, jd)	Returns ranked resumes in DataFrame format
main.py	**launch_gui()**	Launches the Tkinter-based GUI
	export_results(df, path)	Saves ranked results to a CSV file

Chapter 6

Data Design

6.1. Data Model

FIELD	TYPE	DESCRIPTION
Filename	String	Name of the uploaded resume file
Text	string	Extracted and cleaned resume content
Score	float	Cosine similarity score with JD

6.2. Input/Output Mechanism

The system accepts text and file inputs from the user. It outputs scores in a sortable table and downloadable CSV.

- Input: job description (text), resume folder (.pdf/.docx).
- Output: Ranked resume list in table, stored as ranked_uploaded_resumes.csv.

6.3. Data Retention Policies

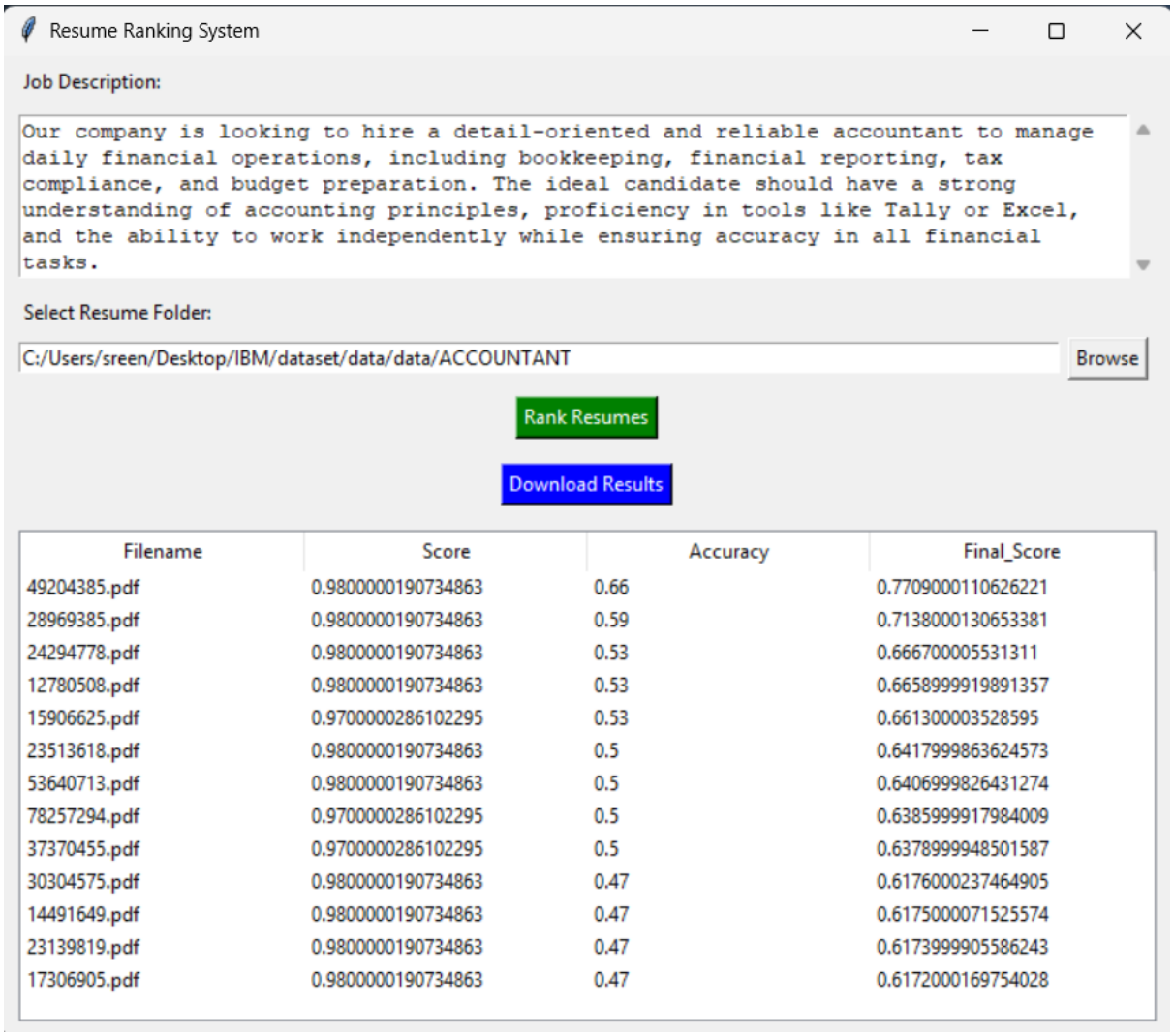
1. Resumes are read from the file system and not saved permanently.
2. Output CSV is only saved for the current session.
3. Personal information is not saved after the program has finished running.

Chapter 7
Interfaces

7.1. User Interface

- Enter job description in multiline text box.
- Click the Browse option to pick the resume folder.
- The rank button starts backend processing.
- The treeview table displays the filename and score.
- Exported CSV for offline viewing.

7.2. Sample Output Interface Screenshot



Filename	Score	Accuracy	Final_Score
49204385.pdf	0.9800000190734863	0.66	0.7709000110626221
28969385.pdf	0.9800000190734863	0.59	0.7138000130653381
24294778.pdf	0.9800000190734863	0.53	0.666700005531311
12780508.pdf	0.9800000190734863	0.53	0.6658999919891357
15906625.pdf	0.9700000286102295	0.53	0.661300003528595
23513618.pdf	0.9800000190734863	0.5	0.6417999863624573
53640713.pdf	0.9800000190734863	0.5	0.6406999826431274
78257294.pdf	0.9700000286102295	0.5	0.6385999917984009
37370455.pdf	0.9700000286102295	0.5	0.6378999948501587
30304575.pdf	0.9800000190734863	0.47	0.6176000237464905
14491649.pdf	0.9800000190734863	0.47	0.6175000071525574
23139819.pdf	0.9800000190734863	0.47	0.6173999905586243
17306905.pdf	0.9800000190734863	0.47	0.6172000169754028

Chapter 8

State and session management

The system functions statelessly, with each application run being isolated. When you quit, all of your user input and results are cleared.

- Every session is separate and isolated.
- There is no session persistence in between application runs.
- Resume and JD data are processed in memory alone.

Chapter 9

Caching

Currently, vector representations are calculated for each session. No results are kept between runs.

- No explicit caching was implemented.
- Vector representations are calculated at runtime for each session.
- Future enhancements could include embedded caching for performance.

Chapter 10

Non-functional requirements

10.1. Security Aspects

Security is provided by confining all processing to the user's local system. There is no data exchange via the network.

- All actions are local, with no external API requests.
- There are no logins, credentials, or sensitive data stored.

10.2. Performance Aspects

The system is designed to work well with small to medium-sized resume datasets. It completes execution in less than 30 seconds for 100 resumes.

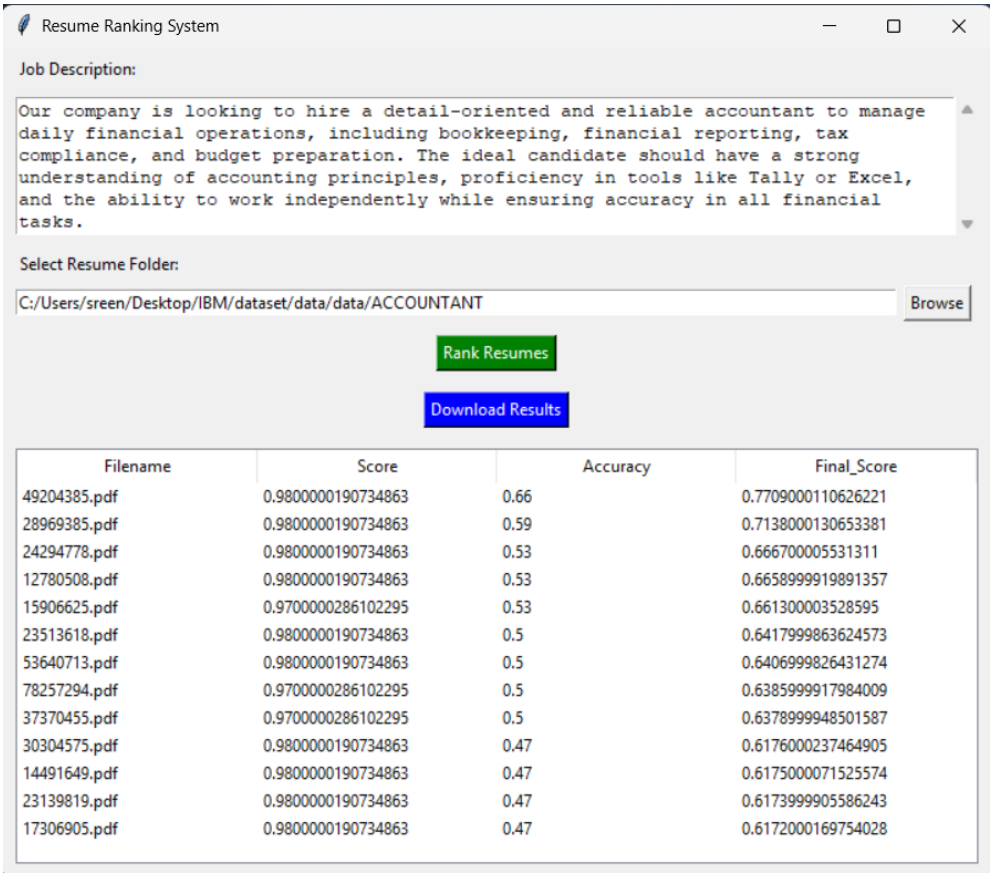
- Processes 100 resumes in ~30 seconds with typical hardware.
- Optimized with NumPy vector stacking and batch processing.

Chapter 11

Screenshots and outputs

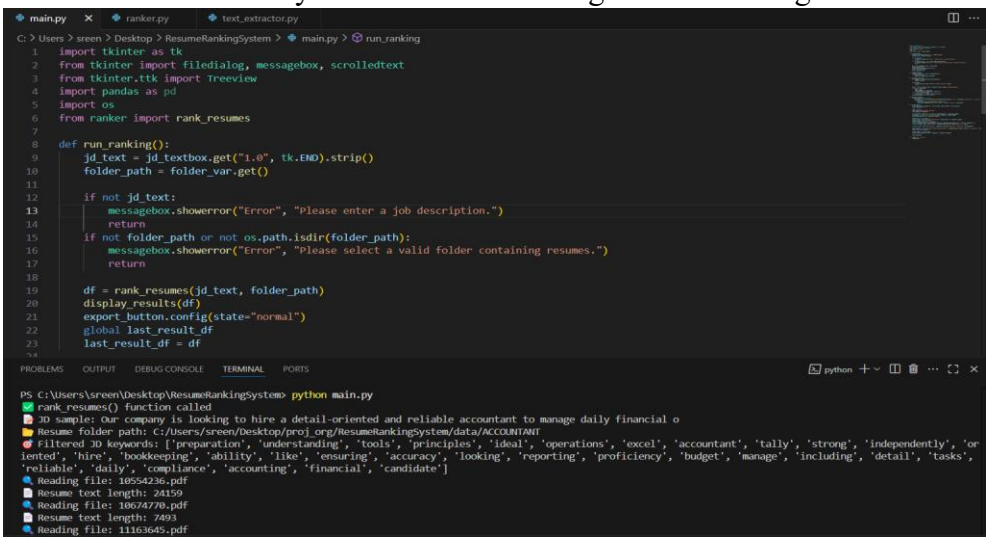
The following screenshots illustrate how the system works:

- Resume Ranking GUI with job description and ranked results:



Filename	Score	Accuracy	Final_Score
49204385.pdf	0.9800000190734863	0.66	0.7709000110626221
28969385.pdf	0.9800000190734863	0.59	0.7138000130653381
24294778.pdf	0.9800000190734863	0.53	0.666700005531311
12780508.pdf	0.9800000190734863	0.53	0.6658999919891357
15906625.pdf	0.9700000286102295	0.53	0.661300003528595
23513618.pdf	0.9800000190734863	0.5	0.6417999863624573
53640713.pdf	0.9800000190734863	0.5	0.6406999826431274
78257294.pdf	0.9700000286102295	0.5	0.6385999917984009
37370455.pdf	0.9700000286102295	0.5	0.6378999948501587
30304575.pdf	0.9800000190734863	0.47	0.6176000237464905
14491649.pdf	0.9800000190734863	0.47	0.6175000071525574
23139819.pdf	0.9800000190734863	0.47	0.6173999905586243
17306905.pdf	0.9800000190734863	0.47	0.6172000169754028

- Code execution and keyword extraction during resume ranking:



Final outputs:

After processing the resumes, the ranked results are displayed in a tabular format based on their relevance to the job description. The final score is computed using a weighted formula combining cosine similarity and keyword match accuracy.

Below is the exported Excel file after ranking:

	A	B	C	D	E	F	G
1	Filename	Score	Accuracy	Final_Score			
2	49204385.	0.98	0.66	0.7709			
3	28969385.	0.98	0.59	0.7138			
4	24294778.	0.98	0.53	0.6667			
5	12780508.	0.98	0.53	0.6659			
6	15906625.	0.97	0.53	0.6613			
7	23513618.	0.98	0.5	0.6418			
8	53640713.	0.98	0.5	0.6407			
9	78257294.	0.97	0.5	0.6386			
10	37370455.	0.97	0.5	0.6379			
11	30304575.	0.98	0.47	0.6176			
12	14491649.	0.98	0.47	0.6175			
13	23139819.	0.98	0.47	0.6174			
14	17306905.	0.98	0.47	0.6172			
15	29821051.	0.98	0.47	0.6165			
16	12802330.	0.98	0.47	0.6164			
17	28298773.	0.97	0.47	0.6159			
18	19446337.	0.97	0.47	0.6157			
19	18132924.	0.97	0.47	0.6151			
20	12065211.	0.97	0.47	0.6145			
21	15289348.	0.95	0.47	0.6053			
22	49997097.	0.98	0.44	0.5953			
23	14449423.	0.98	0.44	0.5942			
24	59403481.	0.98	0.44	0.5939			
25	10554236.	0.98	0.44	0.5935			
26	26975573.	0.98	0.44	0.5928			

Chapter 12

Conclusion

The Resume Ranking System automates resume screening by evaluating their relevance to a job description using NLP techniques such as cosine similarity and keyword accuracy. By combining semantic understanding with keyword relevance, the system provides more reliable results than traditional keyword-only methods.

A simple Tkinter-based GUI enables users to input job descriptions, select resumes (PDF/DOCX), and instantly view ranked results, which can also be exported to CSV. The scoring mechanism prioritizes keyword alignment while incorporating semantic similarity for context.

Overall, the system reduces manual effort, improves hiring efficiency, and remains modular and scalable. With future enhancements like OCR support and BERT-based matching, it holds strong potential for real-world adoption

References

• Explosion AI. (2024). *SpaCy: Industrial-strength Natural Language Processing in Python*. Retrieved from https://spacy.io/

• Scikit-learn Developers. (2024). *Cosine Similarity – scikit-learn documentation*. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.cosine_similarity.html

• PyMuPDF Developers. (2024). *PyMuPDF documentation (fitz)*. Retrieved from https://pymupdf.readthedocs.io/en/latest/

• Shah, A. (2019). *docx2txt: Extract text from docx files*. GitHub Repository. Retrieved from https://github.com/ankushshah89/python-docx2txt

• Python Software Foundation. (2024). *Tkinter documentation*. Retrieved from https://docs.python.org/3/library/tkinter.html

• The pandas development team. (2024). *pandas: Data analysis and manipulation tool*. Retrieved from https://pandas.pydata.org/

• Harris, C. R., et al. (2020). *Array programming with NumPy*. Nature, 585(7825), 357–362. Retrieved from https://numpy.org/

Appendix A

Aspect	Details
Dataset Name	Resume Dataset
Total Records	3446
Resumes with Text	2714 (records containing valid text resumes)
Unique Categories	75 (e.g., Data Science, HR, Engineering, etc.)
Number of Features	169
Feature Names	ID, Resume_str, Resume_html, Category, (others: Unnamed:4 … Unnamed:168)

Appendix B

Code:

```
import tkinter as tk
from tkinter import filedialog, messagebox, scrolledtext
from tkinter.ttk import Treeview
import pandas as pd
import os
from ranker import rank_resumes
def run_ranking():
    jd_text = jd_textbox.get("1.0", tk.END).strip()
    folder_path = folder_var.get()
    if not jd_text:
        messagebox.showerror("Error", "Please enter a job description.")
        return
    if not folder_path or not os.path.isdir(folder_path):
        messagebox.showerror("Error", "Please select a valid folder containing resumes.")
        return
    df = rank_resumes(jd_text, folder_path)
    display_results(df)
    export_button.config(state="normal")
    global last_result_df
    last_result_df = df
def browse_folder():
    folder_selected = filedialog.askdirectory()
    if folder_selected:
```

```
        folder_var.set(folder_selected)
def display_results(df):
    for widget in result_frame.winfo_children():
        widget.destroy()
    if df.empty:
        tk.Label(result_frame, text="No resumes matched.").pack()
        return
    tree = Treeview(result_frame, columns=list(df.columns), show="headings")
    for col in df.columns:
        width = 150
        tree.heading(col, text=col)
        tree.column(col, width=width, anchor="w")
    for _, row in df.iterrows():
        tree.insert("", tk.END, values=list(row))
    tree.pack(expand=True, fill='both')
def export_results():
    if last_result_df is not None:
        save_path = filedialog.asksaveasfilename(defaultextension=".csv", filetypes=[("CSV
files", "*.csv")])
        if save_path:
            last_result_df.to_csv(save_path, index=False)
            messagebox.showinfo("Export Successful", f"Results saved to {save_path}")
def create_gui():
    global jd_textbox, folder_var, result_frame, export_button, last_result_df
    last_result_df = None
    root = tk.Tk()
```

```
    root.title("Resume Ranking System")

    root.geometry("1000x600")

    tk.Label(root, text="Job Description:").pack(anchor="w", padx=10, pady=5)

    jd_textbox = scrolledtext.ScrolledText(root, height=6, wrap=tk.WORD)

    jd_textbox.pack(fill="x", padx=10, pady=5)

    folder_var = tk.StringVar()

    tk.Label(root, text="Select Resume Folder:").pack(anchor="w", padx=10, pady=5)

    folder_frame = tk.Frame(root)

    folder_frame.pack(fill="x", padx=10)

    tk.Entry(folder_frame, textvariable=folder_var, width=60).pack(side="left", fill="x",
expand=True)

    tk.Button(folder_frame, text="Browse", command=browse_folder).pack(side="left",
padx=5)

    tk.Button(root, text="Rank Resumes", command=run_ranking, bg="green",
fg="white").pack(pady=10)

    export_button = tk.Button(root, text="Download Results", command=export_results,
bg="blue", fg="white", state="disabled")

    export_button.pack(pady=5)

    result_frame = tk.Frame(root)

    result_frame.pack(fill="both", expand=True, padx=10, pady=10)

    root.mainloop()

if __name__ == "__main__":

    create_gui()
```

2. ranker.py – Resume Ranking Logic

```
import os

import pandas as pd

from text_extractor import extract_text
```

```
from ranking_engine import clean_text, extract_keywords, match_keywords,
compute_similarity

from spacy.lang.en.stop_words import STOP_WORDS

# Define additional domain-specific generic words to ignore

DOMAIN_STOPWORDS = {

    "company", "organization", "corporation", "work", "employee", "job", "position",

    "skills", "team", "experience", "role", "person", "description", "project"

}

def rank_resumes(jd_text, folder_path):

    print("✅ rank_resumes() function called")

    print("📝 JD sample:", jd_text[:100])

    print("📂 Resume folder path:", folder_path)

    resumes = []

    jd_keywords = extract_keywords(clean_text(jd_text))

    # Filter JD keywords to remove generic or stop words

    important_keywords = [kw for kw in jd_keywords

                          if kw.lower() not in STOP_WORDS

                          and kw.lower() not in DOMAIN_STOPWORDS

                          and len(kw) > 3]

    print("🎯 Filtered JD keywords:", important_keywords)

    for file_name in os.listdir(folder_path):

        if not file_name.lower().endswith((".pdf", ".docx")):

            continue

        file_path = os.path.join(folder_path, file_name)

        print("🔍 Reading file:", file_name)
```

```
        resume_text = extract_text(file_path)
        print("📄 Resume text length:", len(resume_text))
        cleaned_resume = clean_text(resume_text)
        score = compute_similarity(jd_text, resume_text)
        matched, accuracy = match_keywords(resume_text, important_keywords)
        # Normalize and adjust final score (accuracy weighted more)
        norm_score = (score - 0.5) / (1 - 0.5)
        norm_score = max(0, min(1, norm_score))
        boost = 0.05 if len(matched) >= 5 else 0
        final_score = round((0.2 * norm_score) + (0.8 * accuracy) + boost, 4)
        resumes.append({
            "Filename": file_name,
            "Score": round(score, 2),
            "Accuracy": round(accuracy, 2),
            "Final_Score": final_score
        })
    df = pd.DataFrame(resumes)
    print("📈 Final DataFrame shape:", df.shape)
    print(df.head())
    df = df.sort_values(by="Final_Score", ascending=False)
    df.to_csv("ranked_uploaded_resumes.csv", index=False)
    return df
```

3. text_extractor.py – File Parsing

```
import os
```

```
import fitz
# PyMuPDF
import docx2txt
def extract_text_from_pdf(file_path):
    try:
        text = ""
        with fitz.open(file_path) as doc:
            for page in doc:
                text += page.get_text()
        return text
    except:
        return ""
def extract_text_from_docx(file_path):
    try:
        return docx2txt.process(file_path)
    except:
        return ""
def extract_text(file_path):
    ext = os.path.splitext(file_path)[1].lower()
    if ext == ".pdf":
        return extract_text_from_pdf(file_path)
    elif ext == ".docx":
        return extract_text_from_docx(file_path)
    else:
return "
```