Job Match Your Resume

Job Matcher Using Python/Pycharm

Welsey Geer-Laureno,

Dale Quincy, Kenechukwu Agwu,

Steven Roberge

Central Connecticut State University

<u>Table Of Contents</u>

Introduction

The *Resume to Job Match* project aims to create a Python-based application that efficiently connects job seekers with potential employers. With the increasing number of job postings across industries, finding relevant positions that align with a candidate's skills and experiences can be overwhelming. This application uses automation to simplify the job search process by analyzing resumes and matching them to job descriptions that best fit the user's qualifications.

Project Description and Objectives

Project Description

The Resume to Job Match system is designed to read user input (skills, experience, or resume content) and match it with job listings stored in a local database or file. It provides a ranked list of the best matches based on keyword similarities and job requirements.

Objectives:

- To help job seekers find suitable positions faster based on their skill set.
- To help employers identify candidates whose qualifications match their open roles.
- To automate the matching process using keyword comparison or similarity metrics.
- To create a user-friendly interface for both employers and job seekers.

Business Relevance

In a competitive job market, finding the right job or candidate can be time-consuming. This project reduces the manual search effort by intelligently pairing resumes with job openings. With access to a large database (1,000+ jobs), users can instantly find matches that align with their abilities and preferences, improving hiring efficiency and satisfaction for both parties.

Code Documentation and Explanation (Overview)

Libraries Used:

- **Tkinter:** For creating the graphical user interface (GUI).
- **difflib:** For comparing text similarity between resumes and job descriptions.
- **os / csv / json:** To store and manage job listings and resume data.

Planned GUI Components

- **Labels:** To display text and section headers (e.g., "Enter Skills", "Select Job Category").
- **Entry Fields:** For users to input their skills or upload resumes.
- **Buttons:** For actions like "Match Jobs", "Upload Resume", or "Exit".
- **Dropdown Menus:** For selecting job categories or industries.
- **Listbox / Textbox:** To display the top matching job results.

<u>Code Structure Overview:</u>

- `main.py` — main program file that runs the Tkinter window.
- `matcher.py` — contains logic to compare skills and job descriptions.
- `jobs.csv` — stores job listings and requirements.
- `resume.txt` or input field — user resume or skill input.

Instructions for Running the Application

Required Downloads:

- Python 3.8
- Tkinter library
- Resume File (.pdf , .docx , .txt)

How to Use the Application:

- Launch python
- Open application file labeled "job_finder.py"
- Open a terminal prompt window
- Run the Application / GUI
- Click the "Browse" button then locate and select the desired resume file
- A confirmation message will appear after selecting file
- User is prompted to enter/type Job keyword (ex. Server , Teacher)
- User is prompted to select Full-time, Part-Time, or Seasonal checkbox
- User should click the "Find Jobs" button
- To close the Application, User should click "Quit" button

Code Documentation and Explanation (In-depth)

Importing Libraries and Making Core Functionality

The Resume-to-Job Match Project begins by importing a variety of essential Python libraries that make up the foundation of the program. The **Tkinter** library is used to build the graphical user interface (GUI), which provides an accessible way for users to interact with the system (Gaddis, 2024). The GUI incorporates labels, buttons, and text boxes that allow users to upload resumes, enter job keywords, and view matching results in real time. The **difflib** library plays a critical role in measuring the similarity between resumes and job descriptions by comparing textual data and identifying shared keywords. This allows the program to provide a ranked list of jobs based on relevance to the user's input.

In addition to Tkinter and difflib, other standard Python libraries such as **os**, **csv**, and **json** are imported. The os library helps the system locate files and manage directories, while csv and json enable the program to read and write data from job listing files and configuration settings. For example, the csv library allows the program to process large datasets of job postings, each containing attributes such as title, description, and employment type. The json library allows structured data—like saved searches or user preferences—to be stored in a lightweight and easily accessible format. These libraries serve as the backbone of the project, ensuring smooth data handling and reliable file management.

Designing the GUI

The user interface was created using Tkinter and its themed extension, **ttk**, which provides a modernized look to the application's windows and widgets (Python Software Foundation, 2025). Through Tkinter, users are able to click buttons labeled "Upload Resume," "Find Jobs," or "Quit," as well as enter their information into text boxes labeled "Skills" or "Job Keyword." The program also includes an OptionMenu or radio button that allow users to specify whether they are searching for full-time, part-time, or seasonal positions.

The message box feature is another integral part of Tkinter, as it allows the program to display important feedback to the user. For example, if a user attempts to upload a file that is not in a supported format or leaves a required field empty, a warning message will appear. Similarly, when the upload is successful, a confirmation box appears, reinforcing that the user's action was completed correctly. This type of feedback is essential for maintaining usability and preventing errors during the matching process.

Finally, the output window utilizes a **Listbox** or **TreeView** component to display the job matches in a clear, organized way. Each row in the Listbox represents one job posting that the system found to be relevant to the user's uploaded resume or keyword entry. This design allows for an intuitive workflow where users can see the results immediately after submitting their search query, mirroring the responsive experience of modern job search platforms.

Processing Resume Input and Keywords Extraction

Once the user uploads a resume or inputs their skills, the system begins parsing the data using functions defined within the program. A custom function, for example, `def extract_keywords(resume_text):`, is used to separate important terms from the rest of the text. These terms often include words such as "management," "customer service," "Python," or "communication"—depending on the content of the uploaded resume.

The extracted keywords are then stored in a Python **list**, which allows the program to iterate over each keyword and compare it with the words found in the job listings database. During this stage, control structures such as **for** loops and **if/elif/else** statements are used to check for matches. For instance, if a word from the resume appears in a job's description, a similarity counter is incremented to reflect a higher match score.

This process relies heavily on Python's **string methods** and the **difflib.SequenceMatcher()** class, which calculates how closely two strings resemble each other. The higher the similarity ratio, the more closely the resume matches the job description. The results of these comparisons are stored in a **dictionary**, where each job title acts as a key and the calculated similarity score serves as its value. The dictionary is then sorted in descending order, ensuring that the best job matches are displayed first in the GUI.

Implementing Conditional Logic and Control Flow

A major portion of the program's logic is based on **conditional statements** and looping structures that determine how job matches are identified and displayed. After the initial keyword comparison, the system runs a **for loop** through every job listing in the dataset.

Within this loop, multiple **if/elif/else** statements check for specific conditions such as job type, skill match level, or employment category.

For instance, if the user selects “Full-Time” and the job description contains the same label, the program confirms this condition as true and includes it in the output. If no matching category is found, the job is skipped. A **while** loop may also be used to continue checking for matches as long as new job listings are available in the dataset. This ensures that the program can handle large databases efficiently without prematurely ending the search.

Through this layered logic, the Resume-to-Job Match Project demonstrates how Python’s basic programming concepts—loops, conditionals, and lists—can be applied to create a functional and automated job-matching process (Gaddis, 2024).

Ranking and job matching

After all potential matches have been analyzed, the program ranks them using a sorting function based on similarity scores. This is achieved using Python’s built-in `sorted()` method, which sorts the items of the dictionary created earlier. The parameter `reverse=True` ensures that the jobs with the highest similarity scores appear at the top of the list (Gaddis, 2024).

Once sorted, the data is inserted into the Listbox or TreeView component in the Tkinter window. Each job listing is displayed with its title, employment type, and similarity percentage. This gives users a clear and organized view of which jobs best fit their qualifications. The interface also allows users to scroll through results and close the application easily when finished.

Exception handling is used throughout this process to ensure that unexpected errors—such as missing data files or unsupported resume formats—do not crash the program. For example, if a user uploads a file that cannot be read, a message box will appear alerting them to the issue and prompting them to try again. This approach makes the program reliable and user-friendly even for those with limited technical knowledge.

API Use

The code uses an Api key that has yet to be requisitioned. The code plans to for the Api requirement to use the open AI library with the client = OpenAI(key here). The API will be used to connect the function of the code to external job searches. This will give the code practical applications and future longevity

Conclusion

The Resume-to-Job Match Project demonstrates how Python can be used to automate and simplify the process of connecting job seekers with potential employers. By incorporating libraries like Tkinter for interface design, difflib for text comparison, and csv/json for data storage, the program integrates several programming concepts into a cohesive system. Through the use of loops, conditionals, and functions, the application reads resumes, compares skills to job descriptions, and ranks job postings based on similarity.

In the future, this program could be enhanced by connecting to external **job listing APIs**, such as Indeed or LinkedIn, to provide real-time job recommendations. Additionally, integrating **Natural Language Processing (NLP)** would improve the system’s ability to interpret complex job descriptions and resume content. Overall, the Resume-to-Job Match Project showcases the power of programming logic and automation to solve real-world challenges in employment matching (Chandra et al., 2015).

Referneces:

Chandra, R., Gupta, S., & Mishra, P. (2015). *Automated data retrieval using API integration*.

Gaddis, T. (2024). *Starting Out with Python* (6th ed.). Pearson Education.

Python Software Foundation. (2025). *tkinter — Python interface to Tcl/Tk documentation*. https://docs.python.org/3/library/tkinter.html

Part 1 of code (10/17)

```
import tkinter as tk
from tkinter import filedialog, messagebox

class JobFinder:
    def __init__(self):
        # Start the window
        self.window = tk.Tk()
        self.window.title("Simple Job Finder")
        self.window.geometry("600x400")

        # Call setup
        self.make_widgets()

        # Run window
        self.window.mainloop()

    def make_widgets(self):
        # --- Resume upload ---
        tk.Label(self.window, text="Upload your Resume:").grid(row=0, column=0, padx=10,
pady=10, sticky="w")
        self.resume_path = tk.StringVar()
        tk.Entry(self.window, textvariable=self.resume_path, width=40,
state="readonly").grid(row=0, column=1, padx=5)
        tk.Button(self.window, text="Browse", command=self.upload_resume).grid(row=0,
column=2, padx=5)

        # --- Job search box ---
        tk.Label(self.window, text="What job are you looking for?").grid(row=1, column=0,
padx=10, pady=10, sticky="w")
        self.job_entry = tk.Entry(self.window, width=40)
        self.job_entry.grid(row=1, column=1, columnspan=2, padx=5)

        # --- Job type options ---
        tk.Label(self.window, text="Job Type:").grid(row=2, column=0, padx=10, pady=10,
sticky="w")
        self.job_type = tk.StringVar()
        self.job_type.set("Full-Time")  # default

        tk.Radiobutton(self.window, text="Full-Time", variable=self.job_type,
value="Full-Time").grid(row=2, column=1, sticky="w")
        tk.Radiobutton(self.window, text="Part-Time", variable=self.job_type,
value="Part-Time").grid(row=2, column=1, sticky="e")
        tk.Radiobutton(self.window, text="Seasonal", variable=self.job_type,
value="Seasonal").grid(row=2, column=2, sticky="w")

        # --- Buttons ---
        tk.Button(self.window, text="Find Jobs", command=self.find_jobs, bg="green",
fg="white").grid(row=3, column=1, pady=10)
        tk.Button(self.window, text="Quit", command=self.window.quit, bg="red",
fg="white").grid(row=3, column=2, pady=10)

        # --- Job results ---
```

```python
        tk.Label(self.window, text="Matching Jobs:").grid(row=4, column=0, padx=10,
pady=10, sticky="w")
        self.result_box = tk.Listbox(self.window, width=70, height=10)
        self.result_box.grid(row=5, column=0, columnspan=3, padx=10)

    def upload_resume(self):
        try:
            file_path = filedialog.askopenfilename(
                title="Select Resume",
                filetypes=[("PDF files", "*.pdf"), ("Word files", "*.docx"), ("Text
files", "*.txt")]
            )
            if file_path:
                self.resume_path.set(file_path)
                messagebox.showinfo("Upload Complete", "Your resume has been uploaded!")
            else:
                messagebox.showwarning("No File", "You did not pick a file.")
        except Exception as e:
            messagebox.showerror("Error", f"Something went wrong: {e}")

    def find_jobs(self):
        # A simple list of jobs
        jobs = [
            "Cashier - Part-Time",
            "Retail Worker - Seasonal",
            "Software Engineer - Full-Time",
            "Server - Part-Time",
            "Warehouse Worker - Seasonal",
            "Teacher - Full-Time"
        ]

        # Clear old results
        self.result_box.delete(0, tk.END)

        # Get search info
        keyword = self.job_entry.get().lower()
        job_type = self.job_type.get()

        try:
            # Check if resume is uploaded
            if self.resume_path.get() == "":
                messagebox.showwarning("No Resume", "Please upload your resume first.")
                return

            found_any = False  # to track matches

            # Loop through jobs and match
            for job in jobs:
                if keyword in job.lower() and job_type.lower() in job.lower():
                    self.result_box.insert(tk.END, job)
                    found_any = True

            # If no matches found
            if not found_any:
                self.result_box.insert(tk.END, "No jobs found. Try again!")

        except Exception as e:
            messagebox.showerror("Error", f"Problem finding jobs: {e}")



# Run the program
if __name__ == "__main__":
    try:
        JobFinder()
    except Exception as error:
        print("An error happened:", error)
```

<u>clearly outline what the project will accomplish</u>: **This project will be able to connect employers with possible employees. Likewise, employees to relevant employers that match their skill set on their resume.**
<u>What is the business relevance of the project?</u> : **The relevance of this project is allowing people to find jobs easier due to their skill sets. With a system of 1,000+ jobs. It can be hard to find a specific job that pertains to you and what you have to offer. However, this code/application will allow quicker and more efficient ways to find jobs that suit you and your abilities.**
Discuss plan for implementing a graphical user interface (GUI): what widgets are you planning - to use (label, radio button, check box... : **The code will consist of dropdowns, lists, buttons, and have a visual display for the user.**

What libraries, packages you will be using? : **Tkinter, any other libraries we need that we find necessary throughout our coding process**

<u>Part 2 of Code:</u>

```
"""
Resume-to-Job Match Project with OpenAI Embeddings
--------------------------------------------------


This program demonstrates a GUI application that matches a user's uploaded resume
```

```
against job listings using semantic embeddings (OpenAI) instead of just string matching.

Features:
- Tkinter GUI for uploading resumes and entering job keywords
- Automatic PDF and DOCX resume text extraction
- OpenAI embeddings for semantic similarity comparison
- Cosine similarity ranking
- Filter jobs by employment type
"""


# ----------------------------
# Import Libraries
# ----------------------------
import tkinter as tk
from tkinter import filedialog, messagebox
from openai import OpenAI
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
import os

# PDF/DOCX text extraction
from docx import Document
import PyPDF2

# Initialize OpenAI client
client = OpenAI()  # Make sure OPENAI_API_KEY is set in your environment



# ----------------------------
# Helper Functions
# ----------------------------
def extract_text_from_file(file_path):
    """Extract text from .txt, .pdf, or .docx files."""
    ext = os.path.splitext(file_path)[1].lower()
    text = ""

    try:
        if ext == ".txt":
            with open(file_path, "r", encoding="utf-8") as f:
                text = f.read()
        elif ext == ".pdf":
            with open(file_path, "rb") as f:
                reader = PyPDF2.PdfReader(f)
                for page in reader.pages:
                    text += page.extract_text() + "\n"
        elif ext == ".docx":
            doc = Document(file_path)
            for para in doc.paragraphs:
                text += para.text + "\n"
        else:
            raise ValueError("Unsupported file type. Please upload TXT, PDF, or DOCX.")
    except Exception as e:
        raise RuntimeError(f"Error extracting text from file: {e}")

    return text.strip()


def get_embedding(text, model="text-embedding-3-small"):
    """Generate OpenAI embedding vector for given text."""
    response = client.embeddings.create(
        input=text,
        model=model
    )
    return response.data[0].embedding


def rank_resumes_against_jobs(resume_text, jobs):
    """
    Compare a resume to a list of jobs and rank by semantic similarity.
    Returns a sorted list of tuples: (job_title, job_type, similarity_score)
    """
    resume_embedding = get_embedding(resume_text)
    ranked_jobs = []

    for job in jobs:
        job_embedding = get_embedding(job["description"])
        similarity = cosine_similarity([resume_embedding], [job_embedding])[0][0]
        ranked_jobs.append((job["title"], job["type"], round(float(similarity), 4)))

    ranked_jobs.sort(key=lambda x: x[2], reverse=True)  # Highest similarity first
    return ranked_jobs


# ----------------------------
# GUI Class
# ----------------------------
class JobFinder:
    def __init__(self):
        self.window = tk.Tk()
        self.window.title("Resume-to-Job Match Finder")
        self.window.geometry("750x550")
        self.window.resizable(False, False)
        self.make_widgets()
        self.window.mainloop()

    def make_widgets(self):
        """Build GUI components."""
        # Resume Upload
        tk.Label(self.window, text="Upload Your Resume:").grid(row=0, column=0, padx=10,
pady=10, sticky="w")
        self.resume_path = tk.StringVar()
        tk.Entry(self.window, textvariable=self.resume_path, width=55,
state="readonly").grid(row=0, column=1)
        tk.Button(self.window, text="Browse", command=self.upload_resume).grid(row=0,
column=2)

        # Job Keyword
        tk.Label(self.window, text="Job Keyword:").grid(row=1, column=0, padx=10,
pady=10, sticky="w")
        self.job_entry = tk.Entry(self.window, width=55)
        self.job_entry.grid(row=1, column=1, columnspan=2)

        # Job Type
        tk.Label(self.window, text="Employment Type:").grid(row=2, column=0, padx=10,
pady=10, sticky="w")
        self.job_type = tk.StringVar(value="Full-Time")
        tk.Radiobutton(self.window, text="Full-Time", variable=self.job_type,
value="Full-Time").grid(row=2, column=1, sticky="w")
        tk.Radiobutton(self.window, text="Part-Time", variable=self.job_type,
value="Part-Time").grid(row=2, column=1, sticky="e")
        tk.Radiobutton(self.window, text="Seasonal", variable=self.job_type,
value="Seasonal").grid(row=2, column=2, sticky="w")

        # Buttons
        tk.Button(self.window, text="Find Jobs", command=self.find_jobs, bg="green",
fg="white").grid(row=3, column=1, pady=10)
        tk.Button(self.window, text="Quit", command=self.window.quit, bg="red",
fg="white").grid(row=3, column=2, pady=10)

        # Results
        tk.Label(self.window, text="Matching Job Results:").grid(row=4, column=0,
padx=10, pady=10, sticky="w")
        self.result_box = tk.Listbox(self.window, width=90, height=20)
        self.result_box.grid(row=5, column=0, columnspan=3, padx=10)

    def upload_resume(self):
        """Upload resume file and validate."""
        try:
            file_path = filedialog.askopenfilename(
                title="Select Resume",
                filetypes=[("Text files", "*.txt"), ("PDF files", "*.pdf"), ("Word
files", "*.docx")]
            )
            if file_path:
                self.resume_path.set(file_path)
                messagebox.showinfo("Upload Complete", "Resume uploaded successfully!")
            else:
                messagebox.showwarning("No File Selected", "Please select a resume
file.")
        except Exception as e:
            messagebox.showerror("Error", f"Error uploading resume: {e}")

    def find_jobs(self):
        """Perform semantic job matching."""
        # Sample job listings
        jobs = [
            {"title": "Cashier", "type": "Part-Time", "description": "Retail, customer
service, and cash handling."},
            {"title": "Retail Worker", "type": "Seasonal", "description": "Stocking
shelves, assisting customers."},
            {"title": "Software Engineer", "type": "Full-Time", "description": "Python
programming, debugging, teamwork."},
            {"title": "Server", "type": "Part-Time", "description": "Serving customers
and managing orders."},
            {"title": "Warehouse Worker", "type": "Seasonal", "description": "Inventory
management and lifting heavy boxes."},
            {"title": "Teacher", "type": "Full-Time", "description": "Classroom
management and student engagement."}
        ]

        self.result_box.delete(0, tk.END)

        try:
            if not self.resume_path.get():
                messagebox.showwarning("Missing Resume", "Please upload a resume before
searching.")
                return

            resume_text = extract_text_from_file(self.resume_path.get())
            if not resume_text:
                messagebox.showwarning("Empty Resume", "Uploaded resume contains no
text.")
                return

            # Filter jobs by employment type
            filtered_jobs = [job for job in jobs if job["type"].lower() ==
self.job_type.get().lower()]
```

```
            if not filtered_jobs:
                self.result_box.insert(tk.END, "No jobs found for selected employment
type.")
                return

            # Rank jobs semantically
            ranked_jobs = rank_resumes_against_jobs(resume_text, filtered_jobs)

            for title, j_type, score in ranked_jobs:
                self.result_box.insert(tk.END, f"{title} - {j_type} ({score*100:.2f}%
match)")

        except Exception as e:
            messagebox.showerror("Error", f"An unexpected error occurred: {e}")


# ----------------------------
# Main Execution
# ----------------------------
if __name__ == "__main__":
    try:
        JobFinder()
    except Exception as e:
        print("Error starting the application:", e)
```

Look at this one:

```
"""
Resume-to-Job Match Project
---------------------------
This program demonstrates how Python can automate and simplify the process of connecting
job seekers
with potential employers by comparing a user's uploaded resume and job preferences to
available listings.
```

```python
Key Features:
- Tkinter GUI for user interaction (upload resume, enter job keywords, view matches)
- Use of difflib to compare text similarity between resumes and job descriptions
- Basic data handling using built-in Python libraries (os, csv, json)
- Error handling and user feedback through message boxes

References:
Gaddis, T. (2024). Starting Out with Python (6th ed.). Pearson Education.
Python Software Foundation. (2025). tkinter — Python interface to Tcl/Tk documentation.
Chandra, R., Gupta, S., & Mishra, P. (2015). Automated data retrieval using API
integration.
"""

# ----------------------------
# Importing Libraries
# ----------------------------
import tkinter as tk
from tkinter import filedialog, messagebox
import difflib   # Used for text similarity comparison
import os        # Handles file system operations
import csv       # Used for reading/writing job listing data
import json      # Used for saving or loading user preferences



class JobFinder:
    """
    The JobFinder class represents the main GUI application.

    Responsibilities:
    - Building the Tkinter interface for user interaction.
    - Handling resume uploads and validating user input.
    - Comparing user-entered job keywords with mock job listings.
    - Displaying ranked job results in a Listbox.
    """

    def __init__(self):
        """Initialize the main application window and setup widgets."""
        self.window = tk.Tk()
        self.window.title("Resume-to-Job Match Finder")
        self.window.geometry("650x450")
        self.window.resizable(False, False)

        # Call setup function to build GUI components
        self.make_widgets()
        # Run the Tkinter main loop
        self.window.mainloop()

    # ----------------------------------------------------------
    # GUI DESIGN: Labels, Buttons, Text Entries, and Listboxes
    # ----------------------------------------------------------
    def make_widgets(self):
        """Constructs the GUI components using Tkinter widgets."""

        # Resume Upload Section
        tk.Label(self.window, text="Upload Your Resume:").grid(row=0, column=0, padx=10,
pady=10, sticky="w")
        self.resume_path = tk.StringVar()
        tk.Entry(self.window, textvariable=self.resume_path, width=45,
state="readonly").grid(row=0, column=1, padx=5)
        tk.Button(self.window, text="Browse", command=self.upload_resume).grid(row=0,
column=2, padx=5)

        # Job Search Section
        tk.Label(self.window, text="Job Keyword:").grid(row=1, column=0, padx=10,
pady=10, sticky="w")
        self.job_entry = tk.Entry(self.window, width=45)
        self.job_entry.grid(row=1, column=1, columnspan=2, padx=5)

        # Job Type Options using Radio Buttons
        tk.Label(self.window, text="Employment Type:").grid(row=2, column=0, padx=10,
pady=10, sticky="w")
        self.job_type = tk.StringVar(value="Full-Time")
        tk.Radiobutton(self.window, text="Full-Time", variable=self.job_type,
value="Full-Time").grid(row=2, column=1, sticky="w")
        tk.Radiobutton(self.window, text="Part-Time", variable=self.job_type,
value="Part-Time").grid(row=2, column=1, sticky="e")
        tk.Radiobutton(self.window, text="Seasonal", variable=self.job_type,
value="Seasonal").grid(row=2, column=2, sticky="w")

        # Buttons
        tk.Button(self.window, text="Find Jobs", command=self.find_jobs, bg="green",
fg="white").grid(row=3, column=1, pady=10)
        tk.Button(self.window, text="Quit", command=self.window.quit, bg="red",
fg="white").grid(row=3, column=2, pady=10)

        # Results Display
        tk.Label(self.window, text="Matching Job Results:").grid(row=4, column=0,
padx=10, pady=10, sticky="w")
        self.result_box = tk.Listbox(self.window, width=75, height=12)
        self.result_box.grid(row=5, column=0, columnspan=3, padx=10)
    # ----------------------------------------------------------
    # FILE HANDLING: Resume Upload Functionality
    # ----------------------------------------------------------
    def upload_resume(self):
        """
        Allows the user to upload a resume file.
        Provides validation and user feedback through message boxes.
        """
        try:
            file_path = filedialog.askopenfilename(
                title="Select Resume",
                filetypes=[("PDF files", "*.pdf"), ("Word files", "*.docx"), ("Text
files", "*.txt")]
            )
            if file_path:
                self.resume_path.set(file_path)
                messagebox.showinfo("Upload Complete", "Your resume has been uploaded
successfully!")
            else:
                messagebox.showwarning("No File Selected", "Please select a resume file
to continue.")
        except Exception as e:
            messagebox.showerror("Error", f"An error occurred while uploading: {e}")

    # ----------------------------------------------------------
    # CORE FUNCTIONALITY: Job Matching Logic
    # ----------------------------------------------------------
    def find_jobs(self):
        """
        Simulates the job matching process by comparing user keywords with predefined job
listings.
        Uses difflib.SequenceMatcher to calculate similarity ratios and ranks matches.
        """
        # Mock job dataset (could later be loaded from CSV/JSON)
        jobs = [
            {"title": "Cashier", "type": "Part-Time", "description": "Retail, customer
service, and cash handling."},
            {"title": "Retail Worker", "type": "Seasonal", "description": "Stocking
shelves, assisting customers."},
            {"title": "Software Engineer", "type": "Full-Time", "description": "Python
programming, debugging, teamwork."},
            {"title": "Server", "type": "Part-Time", "description": "Serving customers
and managing orders."},
            {"title": "Warehouse Worker", "type": "Seasonal", "description": "Inventory
management and lifting heavy boxes."},
            {"title": "Teacher", "type": "Full-Time", "description": "Classroom
management and student engagement."}
        ]
        # Clear old results
        self.result_box.delete(0, tk.END)

        # Get user input
        keyword = self.job_entry.get().strip().lower()
        job_type = self.job_type.get()

        try:
            # Check if a resume was uploaded
            if not self.resume_path.get():
                messagebox.showwarning("Missing Resume", "Please upload your resume
before searching.")
                return

            # Validate keyword
            if keyword == "":
                messagebox.showwarning("Missing Keyword", "Please enter at least one job
keyword.")
                return

            # Perform job matching using difflib similarity ratios
            ranked_jobs = {}
            for job in jobs:
                if job_type.lower() == job["type"].lower():  # match by employment type
                    similarity = difflib.SequenceMatcher(None, keyword,
job["description"].lower()).ratio()
                    ranked_jobs[job["title"]] = round(similarity * 100, 2)

            # Sort jobs by similarity score (descending)
            sorted_jobs = sorted(ranked_jobs.items(), key=lambda x: x[1], reverse=True)

            # Display results
            if sorted_jobs:
                for title, score in sorted_jobs:
                    self.result_box.insert(tk.END, f"{title} - {job_type} ({score}%
match)")
            else:
                self.result_box.insert(tk.END, "No matching jobs found. Try adjusting
your keyword or job type.")

        except Exception as e:
            messagebox.showerror("Error", f"An unexpected error occurred: {e}")



# ----------------------------------------------------------
# MAIN PROGRAM EXECUTION
# ----------------------------------------------------------
if __name__ == "__main__":
    """
    The main entry point of the program.
   Initializes the JobFinder GUI application and handles any critical startup exceptions.
    """
    try:
        JobFinder()
    except Exception as error:
        print("An error occurred while starting the application:", error)
```

API Code:

```
OPENAI_API_KEY="sk-proj-8l2oUzN50kUNplsFNLkq_h_wEuWS8eU65fsBBulv-
tUvut5Wgaj6KyaWGno8zGnz5gWrDHLviYT3BlbkFJxjzRXGDHImmshqoi11twy-
KGCLIEd3RverkE7p2ZUqUoBUb5qzX8rxlhflWbpztHiME2eBU4UA"
```

Library:

```
dotenv==0.9.9
langchain_openai==1.0.3
chromadb==1.3.4
scikit_learn==1.7.2
numpy==2.3.5
```

Running BackEnd Api Cod:

```
import os
from dotenv import load_dotenv
from langchain_openai import OpenAIEmbeddings

import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

load_dotenv()

nurse_description = """Providing resident-focused, high-quality care while working as a
RN in post-acute facilities
Developing individualized care plans for residents in coordination with the wider medical
team
Reviewing care options with clinicians, nursing staff, residents, and their family
members
Setting up and using medical equipment and devices as needed for effective care
Administering medications and providing treatments to residents consistent with their
care plan
```

```
Measuring and monitoring resident vital signs and drawing fluids as needed for labs and
testing
Ensuring accurate documentation of relevant information such as interventions and
responses, vital signs, and resident medical histories Training and supervising nursing
professionals and support staff as needed, consistent with scope of practice limitations
Supporting staff with incident investigations and reporting Being accountable to follow
and adhere to resident care plans Delivering care consistent with all applicable laws,
policies, and standards of care
"""

engineer_description = """
Establish and maintain project documentation procedures for RFIs, RFCs, submittals,
drawings, contracts, and correspondence
Manage the intake, tracking, approval, and distribution of technical documents using
AASHTOWare, COMPASS, and other platforms
Prepare daily and weekly progress reports, meeting minutes, and change order logs
Coordinate and communicate with resident engineers, contractors, and client
representatives to resolve and document construction issues
Support pay estimates and schedule updates in accordance with CTDOT procedures
Maintain accurate project files and archives for audit readiness
"""

nurse_resume = "Compassionate and detail-oriented Registered Nurse with 6 years of
experience delivering high-quality patient care in acute care and medical-surgical
environments. Skilled in patient assessment, medication administration, care planning,
and interdisciplinary teamwork. Patient assessment and triage; medication administration;
electronic health records; care planning; wound care; IV therapy; patient and family
education; infection control; critical thinking; team collaboration. Professional
Experience: Registered Nurse at Mountain View Medical Center, Denver, CO, March 2020–
Present: Provided nursing care to medical-surgical patients, performed assessments,
monitored vitals, administered medications, collaborated with care teams, educated
patients and families, and maintained accurate documentation using Epic. Previous RN role
at St. Anne’s Regional Hospital, Aurora, CO, June 2018–February 2020: Delivered direct
patient care, assisted with clinical procedures, supported rapid response teams, and
adhered to infection control protocols. Education: Bachelor of Science in Nursing (BSN)
from University of Northern Colorado, Greeley, CO, 2018. Licenses and Certifications:
Registered Nurse (State of Colorado, License #RN472893), BLS, ACLS, PALS, and a specialty
certification in Medical-Surgical Nursing (CMSRN). Professional Affiliations: American
Nurses Association and Colorado Nurses Association. Additional Information: Languages:
English and Spanish; technical skills include Epic, Cerner, infusion pumps, and vital
sign monitoring equipment; volunteer experience with Denver Community Health Outreach."

job_descriptions = [nurse_description, engineer_description]

# Create teh vector embeddings here for the job(s) and the resume input
embeddings = OpenAIEmbeddings(model="text-embedding-3-large", api_key="sk-proj-
8l2oUzN50kUNplsFNLkq_h_wEuWS8eU65fsBBulv-
tUvut5Wgaj6KyaWGno8zGnz5gWrDHLviYT3BlbkFJxjzRXGDHImmshqoi11twy-
```

```python
KGCLIEd3RverkE7p2ZUqUoBUb5qzX8rxlhflWbpztHiME2eBU4UA")

# job embeddings
job_embeddings = []

for job_description in job_descriptions:
    job_embedding = embeddings.embed_query(job_description)
    job_embeddings.append(job_embedding)

# hardcoded examples use for loop above
# nurse_job_embedding = embeddings.embed_query(nurse_description)
# engineer_job_embedding = embeddings.embed_query(engineer_description)

#resume embedding(s)
resume_embedding = embeddings.embed_query(nurse_resume)

#convert the embeddings to a numpy array
embedding_arrays = []
for job_embedding in job_embeddings:
    embedding_arrays.append(np.array(job_embedding))

# nurse_job_arr = np.array(nurse_job_embedding)
# engineer_job_arr = np.array(engineer_job_embedding)

resume_embedding_array = np.array(resume_embedding)



#run cosine similarity on the job description arrays and the resume_embedding_array
job_similarities = []

for embedding_array in embedding_arrays:
    job_similarities.append(cosine_similarity(embedding_array.reshape(1, -1),
resume_embedding_array.reshape(1, -1))[0][0])

for job_similarity in job_similarities:
    print(f"Job Similarity: {job_similarity}")

# nurse_similarity = cosine_similarity(nurse_job_arr.reshape(1, -1),
nurse_resume_arr.reshape(1, -1))[0][0]
# engineer_similarity = cosine_similarity(engineer_job_arr.reshape(1, -1),
nurse_resume_arr.reshape(1, -1))[0][0]

# print(f"Engineer Similarity: {engineer_similarity}")
# print(f" Nurse Similarity: {nurse_similarity}")
# print(f"Length of embedding: {len(job_embedding)}")
```

One Page Summary

Our updated version of the original project introduces several major enhancements to the GUI interface and integrates the OpenAI Embeddings API to significantly improve accuracy, performance, and user experience. These improvements transform the application from a simple keyword-matching tool into an intelligent, AI-driven resume-to-job matching system.

Key Accomplishments and New Features

- **Integrated OpenAI Embeddings API** to analyze resumes and compare them directly to job descriptions, allowing for deeper semantic understanding rather than basic keyword scanning.

- **Implemented cosine similarity scoring** to measure match quality, enabling the program to identify the closest resume-job relationships with much higher precision.
- **Updated the GUI to display the top two best job matches**, each shown with clear percentage match scores for easy interpretation.
- **Removed outdated job-type filter options** (full-time, part-time, seasonal) to streamline the interface and reflect the system's new AI-driven matching process.
- **Added a results listbox** that organizes and displays all job matches for better user visibility and interaction.
- **Introduced error-handling pop-ups** for missing resumes, upload failures, and API-related issues, preventing crashes and improving reliability.
- **Created and tested multiple mock resumes** to verify the accuracy and consistency of the matching algorithm.
- **Included full job descriptions** in the processing pipeline to ensure the AI receives complete contextual information when evaluating resumes.

Overview of Improvements

The previous GUI relied on a simple keyword-matching approach that often produced limited or inaccurate results. With the integration of modern AI tools, the new application can understand context, interpret semantic meaning, and produce more reliable matches. These upgrades significantly enhance usability, accuracy, and overall functionality, making the system more useful for a wide range of users.

Challenges Encountered

Throughout development, we encountered several issues that required debugging and redesigning:

- Ensuring the code could successfully read and process multiple file formats including **PDF, DOCX, and TXT**.
- Troubleshooting **API failures caused by malformed or invalid API keys**.
- Managing **error pop-ups triggered by incorrect resume uploads** or missing files.

Despite these challenges, each issue led to stronger error handling and a more stable final product.

Type One

mock_engineer_resume

```
import tkinter as tk
from tkinter import filedialog, messagebox
import numpy as np
from langchain_openai import OpenAIEmbeddings
from sklearn.metrics.pairwise import cosine_similarity

OPENAI_API_KEY= "sk-proj-8l2oUzN50kUNplsFNLkq_h_wEuWS8eU65fsBBulv-
tUvut5Wgaj6KyaWGno8zGnz5gWrDHLviYT3BlbkFJxjzRXGDHImmshqoi11twy-
KGCLIEd3RverkE7p2ZUqUoBUb5qzX8rxlhflWbpztHiME2eBU4UA"



# ------------------------------
# Job Descriptions
# ------------------------------
nurse_description = """Providing resident-focused, high-quality care while working as a
```

```
RN in post-acute facilities
Developing individualized care plans for residents in coordination with the wider medical
team
Reviewing care options with clinicians, nursing staff, residents, and their family
members
Setting up and using medical equipment and devices as needed for effective care
Administering medications and providing treatments to residents consistent with their
care plan
Measuring and monitoring resident vital signs and drawing fluids as needed for labs and
testing
Ensuring accurate documentation of relevant information such as interventions and
responses, vital signs, and resident medical histories Training and supervising nursing
professionals and support staff as needed, consistent with scope of practice limitations
Supporting staff with incident investigations and reporting Being accountable to follow
and adhere to resident care plans Delivering care consistent with all applicable laws,
policies, and standards of care
"""


engineer_description = """
Establish and maintain project documentation procedures for RFIs, RFCs, submittals,
drawings, contracts, and correspondence
Manage the intake, tracking, approval, and distribution of technical documents using
AASHTOWare, COMPASS, and other platforms
Prepare daily and weekly progress reports, meeting minutes, and change order logs
Coordinate and communicate with resident engineers, contractors, and client
representatives to resolve and document construction issues
Support pay estimates and schedule updates in accordance with CTDOT procedures
Maintain accurate project files and archives for audit readiness
"""


job_descriptions = [("Nurse Job", nurse_description), ("Engineer Job",
engineer_description)]

# ------------------------------
# GUI Application
# ------------------------------
class JobFinder:
    def __init__(self):
        self.window = tk.Tk()
        self.window.title("Simple Job Finder")
        self.window.geometry("600x400")


        self.make_widgets()
        self.window.mainloop()


    def make_widgets(self):
        tk.Label(self.window, text="Upload your Resume:").grid(row=0, column=0, padx=10,
pady=10, sticky="w")
```

```python
        self.resume_path = tk.StringVar()
        tk.Entry(self.window, textvariable=self.resume_path, width=40,
state="readonly").grid(row=0, column=1, padx=5)
        tk.Button(self.window, text="Browse", command=self.upload_resume).grid(row=0,
column=2, padx=5)

        tk.Label(self.window, text="Keyword (optional):").grid(row=1, column=0, padx=10,
pady=10, sticky="w")
        self.job_entry = tk.Entry(self.window, width=40)
        self.job_entry.grid(row=1, column=1, columnspan=2, padx=5)

        tk.Button(self.window, text="Find Jobs", command=self.find_jobs, bg="green",
fg="white").grid(row=2, column=1, pady=10)
        tk.Button(self.window, text="Quit", command=self.window.quit, bg="red",
fg="white").grid(row=2, column=2, pady=10)

        tk.Label(self.window, text="Matching Jobs:").grid(row=3, column=0, padx=10,
pady=10, sticky="w")
        self.result_box = tk.Listbox(self.window, width=70, height=10)
        self.result_box.grid(row=4, column=0, columnspan=3, padx=10)

    def upload_resume(self):
        try:
            file_path = filedialog.askopenfilename(
                title="Select Resume",
                filetypes=[
                    ("All Files", "*.*"),
                    ("PDF files", "*.pdf"),
                    ("Word files", "*.docx"),
                    ("Text files", "*.txt")
                ]
            )
            if file_path:
                self.resume_path.set(file_path)
                messagebox.showinfo("Upload Complete", "Your resume has been uploaded!")
            else:
                messagebox.showwarning("No File", "You did not pick a file.")
        except Exception as e:
            messagebox.showerror("Error", f"Something went wrong: {e}")

    def find_jobs(self):
        self.result_box.delete(0, tk.END)

        if self.resume_path.get() == "":
            messagebox.showwarning("No Resume", "Please upload your resume first.")
            return
        try:
            # Read resume text (simple example, only .txt)
            resume_file = self.resume_path.get()
            if resume_file.endswith(".txt"):
                with open(resume_file, "r", encoding="utf-8") as f:
                    resume_text = f.read()
            else:
                resume_text = "Sample resume text placeholder"  # For PDFs/DOCX
(simplified)

            # Generate embeddings
            embeddings = OpenAIEmbeddings(model="text-embedding-3-large",
api_key="YOUR_API_KEY_HERE")

            # Job embeddings
            job_embeddings = [np.array(embeddings.embed_query(desc)) for _, desc in
job_descriptions]

            # Resume embedding
            resume_embedding = np.array(embeddings.embed_query(resume_text))

            # Compute similarities
            similarities = [cosine_similarity(job_embeddings[i].reshape(1, -1),
                                              resume_embedding.reshape(1, -1))[0][0]
                            for i in range(len(job_descriptions))]

            # Prepare results as percentages
            results = [(round(sim*100, 2), name) for sim, (name, _) in zip(similarities,
job_descriptions)]
            results.sort(reverse=True)

            # Display top two matches
            for percent, name in results[:2]:
                self.result_box.insert(tk.END, f"{name}: {percent}% match")
            self.result_box.insert(tk.END, f"\nBest Match: {results[0][1]}")

        except Exception as e:
            messagebox.showerror("Error", f"Problem finding jobs: {e}")

# ------------------------------
# Run the program
# ------------------------------
if __name__ == "__main__":
    try:
        JobFinder()
    except Exception as error:
        print("An error happened:", error)
```

Type Two

```
"""
Resume-to-Job Match Ranking System
----------------------------------
This script:

1. Creates embeddings for job descriptions and resumes.
2. Computes cosine similarity between them.
3. Ranks resumes for each job posting.
4. Prints the results in a clean table.

Dependencies:
    pip install langchain-openai scikit-learn numpy python-dotenv
"""

import os
```

```
from dotenv import load_dotenv
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
from langchain_openai import OpenAIEmbeddings

# ----------------------------------------------------------
#  LOAD API KEY
# ----------------------------------------------------------
load_dotenv()
API_KEY = os.getenv("sk-proj-8l2oUzN50kUNplsFNLkq_h_wEuWS8eU65fsBBulv-
tUvut5Wgaj6KyaWGno8zGnz5gWrDHLviYT3BlbkFJxjzRXGDHImmshqoi11twy-
KGCLIEd3RverkE7p2ZUqUoBUb5qzX8rxlhflWbpztHiME2eBU4UA")




embeddings_model = OpenAIEmbeddings(
    model="text-embedding-3-large",
    api_key="sk-proj-8l2oUzN50kUNplsFNLkq_h_wEuWS8eU65fsBBulv-
tUvut5Wgaj6KyaWGno8zGnz5gWrDHLviYT3BlbkFJxjzRXGDHImmshqoi11twy-
KGCLIEd3RverkE7p2ZUqUoBUb5qzX8rxlhflWbpztHiME2eBU4UA"
)

# ----------------------------------------------------------
#  SAMPLE JOB DESCRIPTIONS
# ----------------------------------------------------------
job_descriptions = {
    "Registered Nurse (RN)": """
Provide high-quality clinical care; administer medications; monitor vital signs;
develop care plans; collaborate with multidisciplinary teams; maintain accurate
documentation.
""",

    "Construction Project Engineer": """
Manage RFIs, submittals, and documentation; coordinate with contractors; track project
progress;
prepare reports; support schedule updates; maintain accurate project records.
""",
}

# ----------------------------------------------------------
#  SAMPLE RESUMES (3)
# ----------------------------------------------------------
resumes = {
    "Resume A - Nurse": """
Registered Nurse with 5 years in acute care. Skilled in patient assessment, medication
administration, IV therapy, wound care, interdisciplinary collaboration, and EMR systems
like Epic.
""",

    "Resume B - Project Engineer": """
Civil engineer with experience managing RFIs, submittals, construction documentation,
meeting minutes, site inspections, and contractor coordination using AASHTOWare and
Procore.
""",

    "Resume C - Retail Associate": """
Retail professional with customer service experience, cash handling, stocking, loss
prevention,
and POS system operation. Strong communication and teamwork.
"""
}

# -----------------------------------------------------------
#  EMBED JOBS + RESUMES
# -----------------------------------------------------------
print("\nGenerating embeddings...")

job_vectors = {
    job_title: np.array(embeddings_model.embed_query(description))
    for job_title, description in job_descriptions.items()
}

resume_vectors = {
    resume_name: np.array(embeddings_model.embed_query(text))
    for resume_name, text in resumes.items()
}

# -----------------------------------------------------------
#  COMPUTE SIMILARITY + RANK RESUMES
# -----------------------------------------------------------
def rank_resumes_for_job(job_title, job_vec):
    """Return a sorted list of (resume, score)."""
    results = []

    for resume_name, resume_vec in resume_vectors.items():
        similarity = cosine_similarity(
            job_vec.reshape(1, -1),
            resume_vec.reshape(1, -1)
        )[0][0]

        results.append((resume_name, float(similarity)))

    return sorted(results, key=lambda x: x[1], reverse=True)
```

```
# ------------------------------------------------------------
#  OUTPUT RANKED RESULTS
# ------------------------------------------------------------
print("\n==============================")
print(" RESUME MATCH RESULTS")
print("==============================\n")

for job_title, job_vec in job_vectors.items():
    print(f"\n ◇  Job Posting: {job_title}")
    print("-" * 50)

    ranked = rank_resumes_for_job(job_title, job_vec)

    for resume_name, score in ranked:
        print(f"{resume_name:<30}  →  Similarity: {score:.4f}")

    print("\n")


print("\nDone!")

"""
Resume-to-Job Match Project
---------------------------
This program demonstrates how Python can automate and simplify the process of connecting
job seekers
with potential employers by comparing a user's uploaded resume and job preferences to
available listings.

Key Features:
- Tkinter GUI for user interaction (upload resume, enter job keywords, view matches)
- Use of difflib to compare text similarity between resumes and job descriptions
- Basic data handling using built-in Python libraries (os, csv, json)
- Error handling and user feedback through message boxes

References:
Gaddis, T. (2024). Starting Out with Python (6th ed.). Pearson Education.
Python Software Foundation. (2025). tkinter — Python interface to Tcl/Tk documentation.
Chandra, R., Gupta, S., & Mishra, P. (2015). Automated data retrieval using API
integration.
"""


# ----------------------------
# Importing Libraries
# ----------------------------
```

```
import tkinter as tk
from tkinter import filedialog, messagebox
import difflib   # Used for text similarity comparison
import os        # Handles file system operations
import csv       # Used for reading/writing job listing data
import json      # Used for saving or loading user preferences



class JobFinder:
    """
    The JobFinder class represents the main GUI application.

    Responsibilities:
    - Building the Tkinter interface for user interaction.
    - Handling resume uploads and validating user input.
    - Comparing user-entered job keywords with mock job listings.
    - Displaying ranked job results in a Listbox.
    """

    def __init__(self):
        """Initialize the main application window and setup widgets."""
        self.window = tk.Tk()
        self.window.title("Resume-to-Job Match Finder")
        self.window.geometry("650x450")
        self.window.resizable(False, False)

        # Call setup function to build GUI components
        self.make_widgets()

        # Run the Tkinter main loop
        self.window.mainloop()

    # -----------------------------------------------------------
    # GUI DESIGN: Labels, Buttons, Text Entries, and Listboxes
    # -----------------------------------------------------------
    def make_widgets(self):
        """Constructs the GUI components using Tkinter widgets."""

        # Resume Upload Section
        tk.Label(self.window, text="Upload Your Resume:").grid(row=0, column=0, padx=10,
pady=10, sticky="w")
        self.resume_path = tk.StringVar()
        tk.Entry(self.window, textvariable=self.resume_path, width=45,
state="readonly").grid(row=0, column=1, padx=5)
        tk.Button(self.window, text="Browse", command=self.upload_resume).grid(row=0,
column=2, padx=5)
```

```python
        # Job Search Section
        tk.Label(self.window, text="Job Keyword:").grid(row=1, column=0, padx=10,
pady=10, sticky="w")
        self.job_entry = tk.Entry(self.window, width=45)
        self.job_entry.grid(row=1, column=1, columnspan=2, padx=5)

    # Buttons
        tk.Button(self.window, text="Find Jobs", command=self.find_jobs, bg="green",
fg="white").grid(row=3, column=1, pady=10)
        tk.Button(self.window, text="Quit", command=self.window.quit, bg="red",
fg="white").grid(row=3, column=2, pady=10)

        # Results Display
        tk.Label(self.window, text="Matching Job Results:").grid(row=4, column=0,
padx=10, pady=10, sticky="w")
        self.result_box = tk.Listbox(self.window, width=75, height=12)
        self.result_box.grid(row=5, column=0, columnspan=3, padx=10)

    # ----------------------------------------------------------
    # FILE HANDLING: Resume Upload Functionality
    # ----------------------------------------------------------
    def upload_resume(self):
        """
        Allows the user to upload a resume file.
        Provides validation and user feedback through message boxes.
        """
        try:
            file_path = filedialog.askopenfilename(
                title="Select Resume",
                filetypes=[("PDF files", "*.pdf"), ("Word files", "*.docx"), ("Text
files", "*.txt")]
            )
            if file_path:
                self.resume_path.set(file_path)
                messagebox.showinfo("Upload Complete", "Your resume has been uploaded
successfully!")
            else:
                messagebox.showwarning("No File Selected", "Please select a resume file
to continue.")
        except Exception as e:
            messagebox.showerror("Error", f"An error occurred while uploading: {e}")

    # ----------------------------------------------------------
    # CORE FUNCTIONALITY: Job Matching Logic
    # ----------------------------------------------------------
    def find_jobs(self):
        """
        Simulates the job matching process by comparing user keywords with predefined job
listings.
        Uses difflib.SequenceMatcher to calculate similarity ratios and ranks matches.
        """
        # Mock job dataset (could later be loaded from CSV/JSON)
        jobs = [
            {"title": "Cashier", "type": "Part-Time", "description": "Retail, customer
service, and cash handling."},
            {"title": "Retail Worker", "type": "Seasonal", "description": "Stocking
shelves, assisting customers."},
            {"title": "Software Engineer", "type": "Full-Time", "description": "Python
programming, debugging, teamwork."},
            {"title": "Server", "type": "Part-Time", "description": "Serving customers
and managing orders."},
            {"title": "Warehouse Worker", "type": "Seasonal", "description": "Inventory
management and lifting heavy boxes."},
            {"title": "Teacher", "type": "Full-Time", "description": "Classroom
management and student engagement."}
        ]

        # Clear old results
        self.result_box.delete(0, tk.END)

        # Get user input
        keyword = self.job_entry.get().strip().lower()
        job_type = self.job_type.get()

        try:
            # Check if a resume was uploaded
            if not self.resume_path.get():
                messagebox.showwarning("Missing Resume", "Please upload your resume
before searching.")
                return

            # Validate keyword
            if keyword == "":
                messagebox.showwarning("Missing Keyword", "Please enter at least one job
keyword.")
                return

            # Perform job matching using difflib similarity ratios
            ranked_jobs = {}
            for job in jobs:
                if job_type.lower() == job["type"].lower():  # match by employment type
                    similarity = difflib.SequenceMatcher(None, keyword,
job["description"].lower()).ratio()
                    ranked_jobs[job["title"]] = round(similarity * 100, 2)
            # Sort jobs by similarity score (descending)
            sorted_jobs = sorted(ranked_jobs.items(), key=lambda x: x[1], reverse=True)

            # Display results
            if sorted_jobs:
                for title, score in sorted_jobs:
                    self.result_box.insert(tk.END, f"{title} - {job_type} ({score}%
match)")
            else:
                self.result_box.insert(tk.END, "No matching jobs found. Try adjusting
your keyword or job type.")

        except Exception as e:
            messagebox.showerror("Error", f"An unexpected error occurred: {e}")


# ------------------------------------------------------------
# MAIN PROGRAM EXECUTION
# ------------------------------------------------------------
if __name__ == "__main__":
    """
    The main entry point of the program.
   Initializes the JobFinder GUI application and handles any critical startup exceptions.
    """
    try:
        JobFinder()
    except Exception as error:
        print("An error occurred while starting the application:", error)
```