Ai Driven Curriculum Vitae Analysis Model

Final Report

Student Id: 4106983
Email: alkandaz@lsbu.ac.uk
BEng (Hons) Electrical and Electronic Engineering
Module Code: EEE_6_PRO

Author’s signature: Zeiad Alkandari

Table of Contents

Abstract. An increasing amount of digital hiring, paired with the growing need for scalable talent assessment, has driven an increasing interest in intelligent and automated curriculum vitae (CV) systems. Traditional manual methods usually are expensive, slow, inconsistent and biased, and are incapable of handling large volumes of applications. While applicant tracking systems (ATS) exist, they lack the ability to understand context as clearly as they should, being relevant, and understanding candidate's skills in depth. This project presents an AI driven CV analysis platform that will perform skill extraction, job description (JD) matching, and user engagement through surveys, quizzes and more evaluation. The work is divided into three phases. In the first phase, synthetically generated CVs are analysed to assign users access to specific quizzes, survey's and more, based on their skill profiles using a fuzzy logic enhanced NLP pipeline that will also be used to send CV details to be stored in MongoDB. Phase two allows JD uploads to be compared against the database of synthetic CVs using weighted skill matching algorithms. Phase three introduces a user upload interface for both CVs and JDs, using a sentence transformer based semantic similarity scoring to rank matches paired with skill matching algorithm that is also used in Phase 1 and 2. All phases are fully integrated into a web application backed by MongoDB (with the exception of phase 3) architecture and demonstrate skill interpretability using the ESCO API. Across all stages both accuracy and user experience are preserved by employing preprocessing, fuzzy keyword expansion and OCR enabled parsing for a more robust performance on various document formats. Experiments are conducted with synthetically generated data with a predefined criteria for CVs and JDs, validating the systems matching precision using cosine similarity and weighted relevance metrics. The final model successfully bridges the gap between unstructured CV data and structured job role expectations, offering scalable, privacy preserving career matching for future platforms.

1 Introduction

In recent years, there have been considerable interest in the development of intelligent, automated tools for the use of talent acquisition and CV analysis [1]. As organisations increasingly shift toward digital recruitment strategies, the demand for scalable, unbiased, and accurate assessment of candidate credentials has grown significantly [2]. While traditional recruitment methods rely on human evaluation, these processes have proven to be often slow, inconsistent, and highly prone to bias especially when processing large volumes of applications [3]. To address this, various solutions such as ATS have come out [4]. However, many of the ATS platforms remain very limited in their ability to understand contextual relevance, identify transferable skills, or interpret nuanced information in unstructured CV formats [5].

Artificial intelligence (AI), and more specifically its subset Natural Language Processing (NLP), presents a compelling opportunity to improve CV analysis through semantic understanding and data driven insights [6]. By leveraging pre trained transformer models and using fuzzy logic, CV data can be meaningfully compared with job requirements, improving both the speed and the depth of applicant's evaluation [7]. Regardless, of these challenges will persist. CVs often contain inconsistently formatted information, embedded visuals, and varied terminology for similar concepts. Furthermore, scalable storage, skill interpretation, and efficient matching algorithms are vital for real world deployment of such systems [8].

This project introduces an AI driven CV analysis model structured into three integrated phases. Phase one focuses on extracting and analysing skills from synthetically generated CVs to match candidates with quizzes surveys and more. Phase two allows JDs to be uploaded and matched against synthetically generated CVs that are stored in the MongoDB database form phase one in a structured manner using a weighted skill comparison algorithm. Phase three enables real users to upload their own CV and JDs, where a sentence transformer-based similarity model ranks the most suitable matches. A MongoDB database backend, OCR enhanced parsing, and the ESCO skill classification API are used throughout to enhance data extraction and skill accuracy. This study will be focused heavily on design and the performance of each phase, demonstrating how AI and NLP can be applied to help in bridging the gap between unstructured CV data and structured job expectations, with an emphasis on scalability, user privacy and interpretability and keeping the model unbiased.

2 Aim/Objectives and Scope of the Project

2.1 Aim

The aim of this project is to develop, a multi-phase AI driven Curriculum Vitae analysis platform capable for skill extraction, job matching, and intelligent user engagement. The system is designed to operate on synthetically generated and user uploaded CVs and JDs, applying NLP methods to assess candidate suitability for specific roles and assign them relevant evaluations through survey's, quizzes and more.

The primary aim here included but is not limited to the development of a web-based application with a backend to support matching logic, data storage, and semantic analysis, all while complying with data privacy considerations set out by the General Data Protection Regulation (GDPR) through the use of synthetic data.

Furthermore, discussion with my industrial supervisor and academic supervisor led to the inclusion of backend infrastructure that uses MongoDB, OCR based CV processing for pdf and docx for scanned documents which provides greater file format flexibility, and dynamic job skill mapping using the ESCO API and much more. The inclusion of fuzzy logic and sentence transformer models were also considered very necessary for a more advanced skill matching and semantic similarity evaluation.

2.2 Objectives

1. Design and implementation of multi-phase AI architecture
    - Development of a backend using pythons flask library, which is capable of parsing, storing, and analysing uploaded CVs and JDs.
    - Implementation of OCR based text extraction using the PyMuPDF and Tesseract library for processing structured and unstructured documents
    - Integration of skill extraction modules using fuzzy logic and ESCO API for job specific skill profiling.
    - Use of sentence transformer (all-mpnet-base-v2) to compute semantic similarity between CVs and JDs in Phase 3.
2. Multiphase pipeline development
    - Phase 1: Analyse synthetic CVs and assign users access to surveys, quizzes and more based on the extracted skills using NLP pipelines and MongoDB storage.
    - Phase 2: Upload JDs and compare them with the synthetic CV database using a weighted skill matching algorithm.
    - Phase 3: Allow users to upload their own CVs and JDs, using semantic similarity scoring to rank results and limit displayed CVs based on match relevance.
3. Frontend development and UI integrations
    - Create an interactive web page using HTML, CSS and JavaScript for file uploads, job selection, and displaying results.
    - Design a user-friendly interface for editing extracted skills and contact information prior to final submission in Phase 1
    - Display JD CV match results with similarity scores and relevant skill list in a interpretable table.
4. System testing, validation, and accuracy analysis
    - Test skill extraction and matching modules using a corpus of synthetically generated CVs and JDs.
    - Validate semantic scoring using cosine similarity metrics and skill weighting logic, with accuracy thresholds defined by acceptable match variance.
    - Report and log matching anomalies and performance trends for future optimization
5. Scalability, privacy, and portability
    - Design system components to allow scaling from the inclusion of more CVs to the database
    - Ensure data privacy through synthetic data use in all testing phases, with consideration for GDPR compliances.
    - Structure backend logic to allow integration into other platforms such as recruitment portals.

2.3 Scope

The scope of this project which is focused on the end-to-end development of a web-based AI powered CV analysis tool through three different phases. The work included the design and implementation of both the backend and frontend architecture, experimental testing with synthetic CVs and JDs, and the application of NLP and AI techniques for skill extraction, semantic similarity comparisons and match ranking.

Development was limited to controlled, pre-defined datasets due to GDPR constraints, though the system is built with extensibility to real user data. Additional features such as recruiter dashboards, interview scheduling tools, and much more advanced skill clustering mechanisms were considered beyond the scope but remain viable for future development iterations.

The system components are modularly designed to allow for adaption of various job markets, industries and AI pipelines.

3 Deliverables

The project aimed to produce the following key deliverables aligned with the above objectives:

- AI CV matching Engine: A fully functioning backend model that supports fuzzy skill extraction, large sector-based list of predefined skill list, JD analysis (apart from phase 1), and CV ranking across the three phases.
- Web Application: An easy-to-use simplistic user interface platform for file uploads, job selection, skill editing, and match result display.
- Database Infrastructure: A MongoDB system to store structured CV data and support real time JD querying.
- Evaluation Dataset and Test Results: Synthetic CVs and JDs, with performance evaluations across matching accuracy, ranking stability, and edge cases.
- Final Report: A comprehensive documentation of system design, methodology, testing, and performance evaluation.
- Presentation Slides: A slide deck for oral presentation and real-life demonstration showing how system works, summarizing system design, objectives, and key outcomes.

4 Technical Background and Context

Previous studies in the field of intelligent recruitment platforms have shown that traditional ATS lack the ability to evaluate unstructured documents effectively and often fail in interpreting the context, semantic relationships, and skill relevance [1-2]. Such limitations pose quite significant barriers when assessing CVs that are not synthetic, which usually include diverse formatting styles, embedded images, scanned content, and non-standard skill terminology. Consequently, recent approaches have turned NLP and semantic similarity models to help in bridging this gap [9].

The AI-driven CV analysis platform developed in this study utilizes sentence transformers, specifically the all-mpnet-base-v2 model to encode the semantic structure of both CVs and JDs [10-11]. A similarity score is then computed using cosine similarity, which is mathematically defined as [11]:

$$Cosine\ Similarity = \frac{\vec{A} \cdot \vec{B}}{\|\vec{A}\| \cdot \|\vec{B}\|} \tag{1}$$

$$where \quad \vec{A} \cdot \vec{B} = \sum_{i=1}^{n} A_i \cdot B_i$$

Where $\vec{A}$ and $\vec{B}$ represent the dense vector embeddings of the CV and JD respectively. This allows the model to compare entire documents based on their semantic content, rather than simply relying on simple surface level keyword matching [11].

To prepare CVs for semantic processing, the system will first extract the text using PyMuPDF for PDFs and the python docx library for word documents. If any content is embedded as images due to it being a scanned pdf or docx document for example, it will be extractable by this method, where optical character recognition (OCR) via the pytesseract library will be automatically applied [12-13]. This would ensure that native and scanned documents are accurately parsed into machine readable text, preserving all skill references and relevant content.

In addition to semantic scoring, fuzzy skill matching will also be incorporated using the **partial_ratio** function from the fuzzy Wuzzy library. This function determines the highest similarity between a query string like a job required skill and any other substring for a longer string like the CV text. It is particularly effective in cases where the skill appears as a part of longer phrase for example, matching “TensorFlow” with “TensorFlow Lite Developer” or something like “java” with “javascript” [14].

Internally, **partial_ratio** uses Pythons SequenceMatcher class, which applies the Ratliff/Obershelp pattern recognition algorithm. The similarity ratio is calculated using:

$$Similarity\ Ratio = \frac{2\ \cdot M}{|a| + |b|} \tag{2}$$

Where M is the number of characters in the longest aligned matching blocks between the input strings a and b and $|a|$ and $|b|$ would be their lengths. The function will evaluate every possible substring of the longer input and return the maximum similarity ratio. This method will offer a much more improved way of prioritizing structural alignment over strict character matching and is less sensitive to small formatting inconsistencies or word order differences [15].

The European Skills, Competences, Qualifications, and Occupations (ESCO) API is employed to automatically get a set of job specific skills by querying an occupations URI [16]. The ESCO ontology supports hierarchical relationships between job roles that are associated with certain competencies, which are stored and reused across the system. When a job role is selected, the ESCO response includes a list of skill clusters, from which the titles are extracted using the regex equation format:

$$\text{titles} \ = \ \text{re.findall}(\text{r}'\text{"title":}\backslash\text{s} * \text{"([\^{}"]+)"}', \text{skill\_response.content.decode}('\text{utf} - 8')) \tag{3}$$

This regular expression searches the raw JSON response returned by the ESCO API and captures al values assigned to the “title” key for further processing.

The system is designed in three integrated phases. In Phase 1, CVs are parsed using PyMuPDF and the python docx library to extract all native text content. If textual data cannot be extracted for example when the content is an embedded image or if there are embedded images OCR via the pytesserat library will be applied using **partial_ratio** function to identify job related skills. This leverages pythons SequenceMatcher, which uses the Ratcliff/Obserhelp algorithm and will compute the similarity according to equation (2). This approach is great at identifying skill mentions in scanned or variability phrased text.

In Phase 2, users can upload job descriptions, which are compared against stored CVs in MongoDB using a weighted sill match score that is defined as:

$$Weighted\ Score = \frac{\sum_{i=1}^{n} w_i\ \cdot\ \delta(s_i)}{\sum_{i=1}^{n} w_i} \tag{4}$$

Where w_i is the importance weight of a skill s_i for a given role, and $\delta(s_i)$ is an indicator function that returns 1 if the skill is found in the CV and 0 otherwise [17].

In Phase 3, users can upload both JDs and CVs simultaneously. A sentence transformer-based comparison pipeline computes semantic similarity scores using equation (1) and fuzzy string comparison scores using equation (2). These are then combined with weighted skill score equation (4) using a weighted average to generate a final ranking. This is defined as:

$$Final\ Score = \ \alpha \cdot Cosine\ Similarity(JD, CV) + (1 - \alpha) \cdot \frac{\sum_{i=1}^{n} w_i\ \cdot\ \delta(s_i)}{\sum_{i=1}^{n} w_i} \tag{5}$$

In addition to similarity scoring, the system logs outcomes, stores results in MongoDB, and offers an interactive web interface. The modular architect enables easy integration of additional analysis tools or external APIs in future development.

Taken together, the mathematical foundation from equation (1-5) and modular pipeline provide a great framework for CV analysis. By combining transformer-based embeddings, pattern based fuzzy logic, and semantic ontologies like ESCO, the system can improve recruitment processing [18-19].

5 Methodology

5.1 Text Extraction

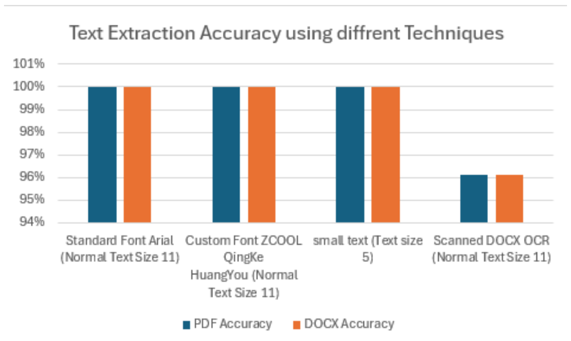
For preprocessing text extraction is the same in all phases where, all input documents CVs or JDs are converted into machine readable text. This system supports both PDF and Word (docx) formats. PDF files are processed using **PyMuPDF** which comes from the **fitz** library and extracts text from each page [12], while docx files are handled via the pythons docx library to read all paragraph contents. To ensure no information is lost, the extractor also incorporates an OCR mechanism for cases where the text appears within images like a scanned documents or in embedded images. For PDFs, each page's images are detected and run through tesseract OCR via the **pytesseract** library [13], and any recognized text is appended to the pages output. Likewise, for docx files, embedded images are pulled and passed to OCR. This two-step approach guarantees that both native text and the text within the images are captured from the CV or JD.

After raw text extraction, a preprocessing step standardizes the content. All characters are lowercased and non-alphanumeric characters like punctuations and special characters are removed, except for whitespace. This normalization which is done via regex manipulation produces a clean, and uniform text string for each document. By removing noise and case differences, the subsequent analysis becomes more robust to varying CV/JD formats. The outcome of this stage is a continuous plain text representation of the documents content, ready for keyword based and semantic processing in the later phases. Notably, the text extraction component is reused throughout the project for every document taken in, ensuring a consistent pipeline for data input in all phases [9,11].

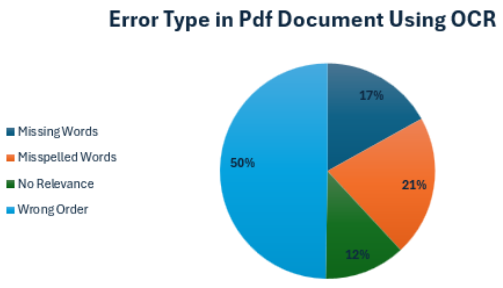
To evaluate the performance of this stage, tests were conducted across three formats standard PDFs, DOCX documents, and scanned images requiring OCR. The results are summarized in Fig. 1 (a), which shows that native PDFs and DOCX files- whether in standard fonts like arial or custom fonts- had a 100% extraction accuracy. However, when dealing with scanned images or text embedded within images, the OCR based extraction introduced errors and reduced the accuracy to 96% [13].

Common issues observed in OCR based extraction for pdfs are detailed in Fig. 1 (b), where 50% of the errors were due to incorrect word ordering and 21% due to misspelled words, typically due to low resolution scans or stylized text layouts [13]. Additional visual examples of these OCR challenges in docx, including OCR misreads and partial extractions from scanned documents, are shown in Fig. 1 (c).

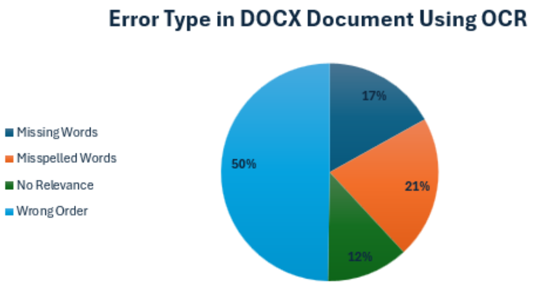
experimental validation demonstrates that while the core extraction system is highly reliable for most standard inputs, OCR related limitations remain active challenge, especially for noisy or non-uniform image data.



(a)



(b)



(c)

Fig 1. (a) Text Extraction Accuracy for different Techniques for Pdf and DOCX, (b) Error Type in PDF document using OCR, (c) Error Type in DOCX document using OCR.

5.2 Predefined Skill List Structure and Selection

A Central component to the CV analysis is a comprehensive and predefined list of skills that will serve as the knowledge base for matching. This list is a collection of technical skills, programming languages, tools, and other keywords that are relevant to a wide range of engineering and IT roles. It is structured by categories for clarity and coverage. For example, it includes programming languages like python, java, C++, JavaScript, SQL, MATLAB frameworks and libraries like react, angular, TensorFlow, pytorch and more and cloud and dev ops tools such as AWS, Azure, Docker, Kubernetes and many more [11]. By organizing the lexicon is thematic groups, we ensured that the list spans over diverse sub domains like machine learning, cloud computing, cybersecurity and more without significant gaps. This predefined lexicon is compiled from domain knowledge and common requirements in job postings, aiming to cover both general skills and niche specific terms.

In addition to these static skill repositories, for phase 1 it also can dynamically tailor the skill list to a given job context using the ESCO API. When a target job title or role is provided by the user for example user types “Data Scientist” as the role of interest, a request is made to ESCOs online database to fetch the most relevant occupation match and its associated skills [16]. The ESCO API returns a set of role specific skills things such as competencies that are expected for that occupation; these are then integrated into our skill list for that session. Before analysis, the retrieved ESCO skills are merged with the predefined list to form an even more enriched list of keywords particularly relevant to the job title in question. This mechanism allows the system to select or emphasize skills that are especially pertinent to the job context being used, without relying solely on the general list. In summary, the skill list used in any analysis is comprehensive and context aware: it contains a broad range of technical terms and can be augmented with occupation specific skills to improve recall of relevant keywords during extraction [11]. This combined lexicon is used in all matching phases that involve identifying skills in text.

5.3 Skill Weighting System

While a list of skills determines what keywords to look for, the skill weighting system determines their relative importance for matching. We designed a weighting scheme to mirror the fact that not all skills are of equal importance for a given job even if they are relevant to a job. For each job role or category, a set of weights is predefined for the relevant skills of that role. These weights are numerical values that typically range from 0.7 up to 2.5 and predefined list skills are 1 if they are present for the role but a manual skill weight is not in place, these skill weights are in place to reflect the significance of each skill: a higher weight means the skills is more crucial for that specific role. For example, in the context of a software engineer role skills like java or JavaScript might be weighted at 1.8-2.0 meaning they are very important, whereas a skill like Docker could have a moderate weight around 0.7 which means its still useful but less central for the role. In a “Data Scientist” role, Python, SQL and TensorFlow might each be given high weights around 2.0-2.5, which would indicate that proficiency in those is essential, while something g like AWS or Tableau might be slightly lower but still significant around 1.5-2.0. Each role in our system has a corresponding dictionary of skill weights reflecting what that role values most. These weight assignments were informed by typical job description emphasis and domain expertise, ensuring that they align with real world expectations for each position.

The skill weight system is used to calculate the match scores between CVs and JDs in a nuanced way. Rather than simply counting the matching skills, the algorithm sums the weights of the specific skills that match. For a particular JD we obtain its required skill set and apply the predefined weights: the total “required weight” is the sum of weights of all skills identified in that JD. When comparing a candidate’s CV to this JD, we sum the weights of the skills that the candidate possesses i.e. the intersection of the CV skills and JD skills. The weight match scored will then be computed as percentage as seen in equation (4).

In equation (4) a score from 0 to 100% would indicate how well the candidates’ skills align with the job requirements, considering the importance. For example, if a job requires {A, B, C} with weights {5, 3, 2} then the sum would {10} and a CV has A and C but not B, the score would be ((5+2)/10) x100% =70%. In this way, missing a high weight skill like in the B example has a larger impact on the score than missing a low weight skill. The weighting system thereby provides a more discriminative comparison metric: candidates strong in the most critical areas will rank higher than those who might match many minor criteria but lack key competencies. This framework which is utilized specifically in Phase 2 ranking of CVs for a given JD, and a modified form is incorporated into the phase 3 similarity scoring. By defining and using this weighted system globally rather than within each phases description, we avoid repetition and ensure consistency in how skill importance is factored into all matching stages.

5.4 Database and Data Storage

In phase 1 of the project, a MongoDB NoSQL database is used persistently store the structured data extracted from each uploaded CV. This allows the system to maintain a repository of CV information things such as skills, qualifications, contact details and more these can be efficiently retrieved later, rather than having to reprocess each CV file on every request. MongoDB is chosen due to its flexible nature in handling semi structured data, as each CVs details can be stored as a document with various fields capturing the applicant's information [17]. All CV documents are kept in a dedicated database (named **cv_analysis2_db** in our case) and collection (named **cvs2**) within MongoDB, creating a centralized storage for Phase 1 outputs. Storing the extracted CV content in this way serves the purpose of enabling the reuse of data in Phase 2 and beyond, while providing persistence across sessions.

Data structure: The content of each CV includes:

Contact Info: Contact details such as email address and phone number, stored under a **contact_info** sub document.

Education: Educational background is extracted from the CV text and is stored in the **education** field.

Experience: A summary of work experience or employment history drawn from the CV, stored in an **experience** field.

Skills: A list of skills identified in the CV and stored in the skills array field.

Years of experience: The total number of years of experience in relevant field, as input by the user, stored in the **years_of_experience** field.

Full CV text: This is the raw text of the entire CV text stored in **cv_text** , in case a issue occurs or there is missing detail this serves as a backup.

CV Hash: Implements a SHA-256 hash of CV text content **cv_hash**, uses a unique fingerprint to detect duplicate uploads.

Filename: The original filename of the CV document, stored for reference and traceability.

Storing a SHA-256 hash of each CVs text is a crucial design choice to prevent duplicate entries of the same CV. When a new CV is uploaded in phase 1, the system computes the hash of its text content and checks the MongoDB collection to see if a document with the same hash already exists. If a match is found, it indicates that a identical CV has already been stored, and the new upload will therefore be recognized as being a duplicate. In such cases, it will not be inserted and will instead be skipped to avoid storing redundant data. This mechanism will ensure data integrity by preventing multiple entries of the same CV. If a hash is unique meaning there is no existing data of that hash, the CV's structured data is augmented with this **cv_hash** and will then be inserted as a new document in the collection. By using the hash as a key for uniqueness, the database can efficiently identify repeats without relying on text comparisons that are expensive [17].

In phase 2, the project capitalizes on the CV data stored in Phase 1 by reusing it for JD to CV matching. The phase 2 backend connects to the same MongoDB database collection to retrieve stored CV entries. This means that the CVs uploaded earlier are readily available for analysis against new JD without requiring the users to reupload their CVs. When a JD is uploaded in phase 2, its text parsed, and relevant skills are extracted. The system then queries the **cvs2** collection for all CV documents and compares each CVs skill set tot the skills that are derived from the JD. For each stored CV, the matching algorithm will cross reference the skills in the CV document with the JD skill requirements to identify common skills and to calculate a match score. Because all CVs are stored in MongoDB, this comparison can iterate through the CV collection and assemble a list of match es efficiently the reuse of Phase data in Phase 3 demonstrates a integrated approach: which is that the database acts as a intermediary, meaning it allows the system to leverage previously stored CV information to find the best candidates for a given JD. This design not only save s time, by helping the system avoiding reprocessing of CV files but also ensures consistency in how C data is used across different phases of the project [17].

5.5 Phase 1: Skill extraction from CVs

Phase 1 focusses on extracting and analysing technical skills from a CV text. The goal is to automatically identify a candidate's skills from their CV and use this information for matching them with the relevant quizzes or evaluation. The skill extraction methodology developed in this phaser uses a hybrid approach where it combines predefined skills with an updated skill list based on the user's job selection which is fetched from the ESCO API [16,22]. Each CV is first parsed (with OCR if necessary) to obtain raw text, which will then be normalized using lowercasing and removing duplicates and will be scanned for occurrences of known skill keywords. A fuzzy string-matching technique using the fuzzy Wuzzy library [14] will be applied so that the system can recognize skills even when there are minor variations in spelling or in phrasing. This fuzzy matching incorporates multiword term handling and length-based penalties to avoid false positives [20]. For example, if a CV contains the skill "machine learnings" where it misspells machine learning, the algorithm can still match it to machine learning due to it having a high similarity to the word we are looking for [14,20]. By leveraging both direct substring matching and fuzzy matching with an adjustable similarity threshold, the system can then capture a wide range of skill mentions, from exact keyword matches to partial and context based matched. All extracted skill candidates from the CV are then cross verified against the ESCO skill list to ensure that they are valid professional skills [16]. The result of this pipeline is a list of identified skills for each CV, which is stored in the database for further use in later phases [17,22].

Initial Evaluation: The skill extraction module was evaluated on a set 15 synthetic CVs. For each CV, the total number of actual skills as listed in the CV was compared against the number of correctly identifiable skills by the system. We also counted wrongly matched skills where the system incorrectly tags a phrase as a skill and missing skills which are skills that are present in the CV but not recognized by the algorithm. The initial implementation of the algorithm achieved a moderate accuracy on average about 21% of the skills in each CV were extracted correctly [22]. In other words, a little under a quarter of each CV's skills were identified accurately by the system, leaving substantial room for improvement. Most CVs had an extraction accuracy in the 10-25% range, and a detailed error analysis revealed that most errors were false negatives [21]. Many of these missing skills were common domain specific terms that were absent from the predefined list, meaning that the algorithm did not know to look for them [20]. There were also false some false positives where the fuzzy matcher inaccurately matched a fragment of text to a skill. This analysis which is summarized in Fig. 2 (a) highlighted that expanding the systems knowledge to skill keywords and fine tuning the matching logic is necessary to boost the accuracy of skill extraction overall [22]. The overall performance at this stage is a proof of concept (POC) and is not sufficient for practical use an accuracy of 21% misses many candidate skills – so the objective here should be to raise the accuracy to at least 70-80% through iterative improvements [21,22].

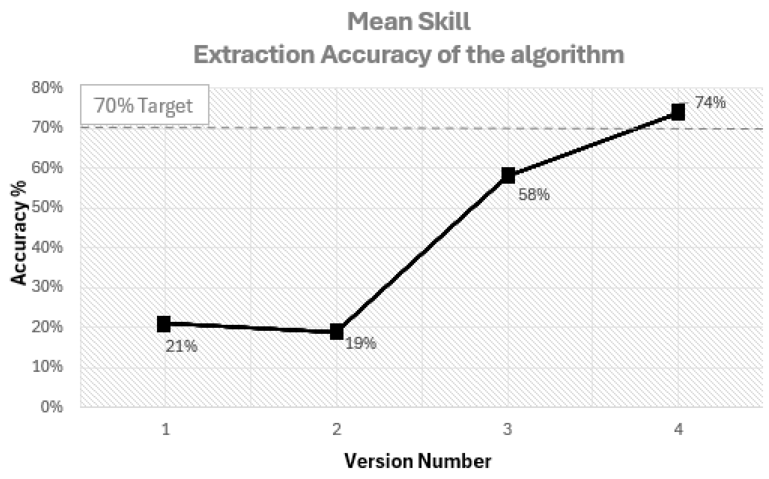
Improvements: To increase the skill extraction accuracy, several enhancements were implemented without fundamentally changing the pipeline structure [22]. The most impactful change was in augmenting the predefined skill list itself. By reviewing the skills that were frequently missed, we identified numerous relevant keywords that were absent from the original list. These missing skills were added to the predefined list so that the system would recognize them in the CV text [22]. Simple expansion of the knowledge base ensued that common skills which previously went unmatched and were counted as missing could now be detected directly. In addition, the fuzzy matching parameters were fine-tuned: the similarity threshold was adjusted, and the matching procedure was made more context aware for multiword skill phrases [14], the algorithm was modified to check each constituent words presence in the CV text using partial ratio matching after the initial fuzzy match, this is done to help reduce false matches [20]. A minor length penalty is also introduced, were very short matches or those with large character differences were discounted unless additional context supported them, which helped cut down on incorrect matches of sub words [14]. These refinements are vital in improving the systems skill extraction accuracy we went through several iterations, summarized as follows:

Version 1: Used a small list of skills and basic text normalization. This approach yielded a skill extraction accuracy of roughly 21% across 15 CVs which would indicate that many skills are missed due to the limited vocabulary [21].

Version 2: Incorporated direct substring matching with a basic fuzzy matching where there is a threshold of 60% similarity for a match. This did not significantly improve the results, the accuracy of these here was around 19% in fact slightly lower which suggests that the threshold needs tuning, and the list is still insufficient [14,20].

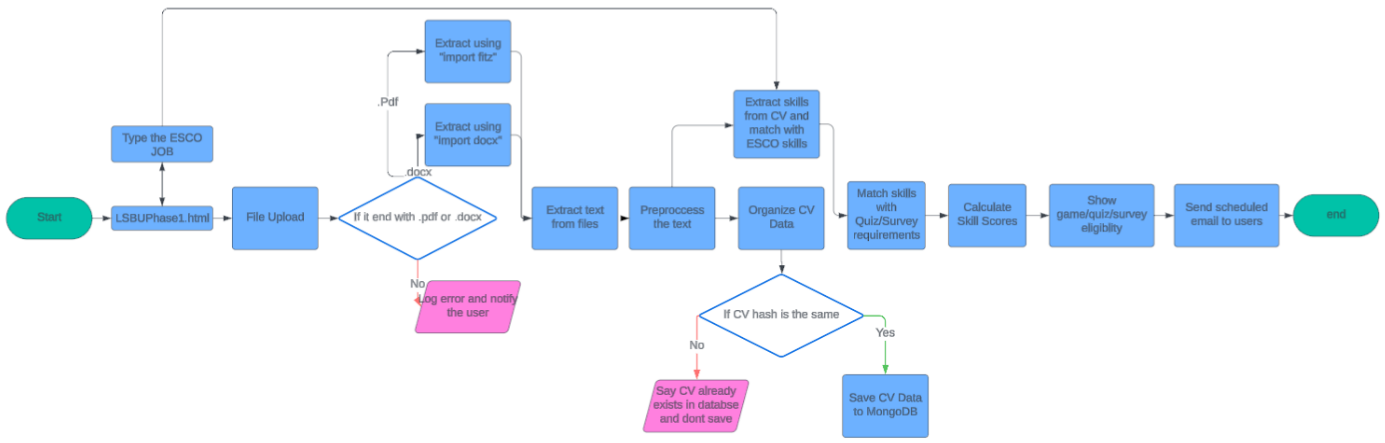
Version 3: Introduced a more advanced fuzzy matching strategy and broader skill list. The version used **fuzz.token_set_ratio** for multi word terms, applied a length-based adjustment and included context checks for multi word skills as described above. These changes dramatically improved the extraction performance, raising the average accuracy to about 58%. Version 3 was able to catch far more skills by recognizing term variations and by also having a larger pool of known skills [14,22].

Version 4: Focused on skill list expansion by adding the most commonly missing skills from prior tests. With broader predefined skill set and the refined fuzzy matching from version 3 kept in place, the systems accuracy jumped to approximately 74%. This surpassed the initial target of 70% accuracy. Notably, this improvement was achieved without any complex new algorithms which ensured that the systems dictionary is more comprehensive and had a major effect on performance [22].



(a)

Fig 2. Mean skill Extraction accuracy of the algorithm based on version


Start
LSBUPhase1.html
Type the ESCO JOB
File Upload
If it end with .pdf or .docx
.Pdf
.docx
Extract using "import fitz"
Extract using "import docx"
No
Log error and notify the user
Extract text from files
Preproccess the text
Extract skills from CV and match with ESCO skills
Organize CV Data
If CV hash is the same
No
Say CV already exists in databse and dont save
Yes
Save CV Data to MongoDB
Match skills with Quiz/Survey requirements
Calculate Skill Scores
Show game/quiz/survey eligiblity
Send scheduled email to users
end

(a)

Fig 3. Phase 1 Final Flowchart

5.6 Phase 2: Weighted Skill Matching with Job descriptions

Phase 2 of the project focuses on matching the skills extracted from candidate CVs to the requirements of a given JD using a weighted skill matching algorithm following equation (4). This phase extends the capabilities from Phase 1 which extracted and normalized skills from CVs by introducing a scoring mechanism that accounts for relative importance of each skill to the job role. The backend implementation for Phase 2 involves parsing the JD to identify relevant skill using a enhanced skill extraction method and then comparing these JD skills against the stored skills in each CV. Each skill is assigned to a predefined weight based on its significance to the role, and these weights directly influence the match score. By weighting critical skills more heavily than ancillary skills, the algorithm can ensure that the candidates that possess the most important qualifications for the job are ranked higher. In summary, Phase 2 creates a role specific skill matching score that is more discriminative and aligned with job requirements than simply basing it on keyword match [23].

Weighted Skill Matching Algorithm Design. To implement this, a skill weighting system was developed. For each job role category, a dictionary of skill weights is defined based on domain knowledge and typical JD emphasis to reflect the importance of each skill for that role. These weights are numeric values typically ranging from roughly 0.7 for less critical skills to up to 2.5. If a skill appears in the general predefined list but has no manually assigned weight for a given role, a default weight of 1 will be used. This scheme mirrors the reality that not all listed skills are equally important: for example, a software engineers JD might assign a high weight like 1.8 to a core programming language for that role like Java or JavaScript, but a lower weight 0.7 to a tool like Docker. Each roles skill weight dictionary encapsulates these priorities, ensuring the matching algorithm focuses on what the job values most [24].

Using these weights, the matching score between a CV and a JD is calculated in a weighted manner rather than a simple count of common skills. First, the JD text is processed to extract the set of required skills. The extraction method builds on Phase 1 text processing techniques: the JD is scanned for any of the known skill keywords from the comprehensive skill list that is used in Phase 1, which includes general and role specific terms. We employed improved normalization and fuzzy matching using the fuzzy wuzzy library to recognize skill mentions even if they appear in varying forms or with minor typos. This was a refinement over the initial implementation early versions of phase 2 relied on basic keyword lookup with a limited list and no fuzzy matching, which missed many relevant terms. By Phase 2 final iteration, the skill extraction uses an expanded skill list and a regex-based matching, supplemented by fuzzy string matching with tuned similarity thresholds to capture partial matches and synonyms [14,25].

Next, for each candidate CV, we take the set of skills that was extracted in Phase 1 the candidates known skills and match it against the set of required skills identified in the JD. This matching process is essentially the intersection between the candidate's skill set and the JDs required skill. However, instead of just counting the number of overlapping skills, the algorithm considers the importance weights associated with each skill. Each JD required skill is assigned a weight as described earlier, and the final score reflects the proportion of these weights that are matched by the candidate [26].

The set of required skills in the JD can be expressed as:

$$S_{JD} = \{s_1, s_2, s_3, \ldots, s_n\} \tag{6}$$

And we let each skill s_i have an associated importance of w_i. Then the total weighted requirement will be defined as:

$$W_{JD} = \sum_{i=1}^{n} w_i \tag{7}$$

Then we let the set of skills present in the candidates CV to be $S_{CV} \subseteq S_{JD}$ i.e. the subset of the JD skills that a user has will be the total weight of matched skills where it would then be defined as:

$$W_{Match} = \sum_{s_i \in S_{CV}} w_i \tag{8}$$

Therefore, this can then be written as the percentage of JDs total skill weight that is covered by the candidates matched skills:

$$Weighted\ Score = \left(\frac{W_{Match}}{W_{JD}}\right) \cdot 100\% \tag{9}$$

For example, a JD requires 3 skills:

Skill A with a weight of 5

Skill B with a weight of 4

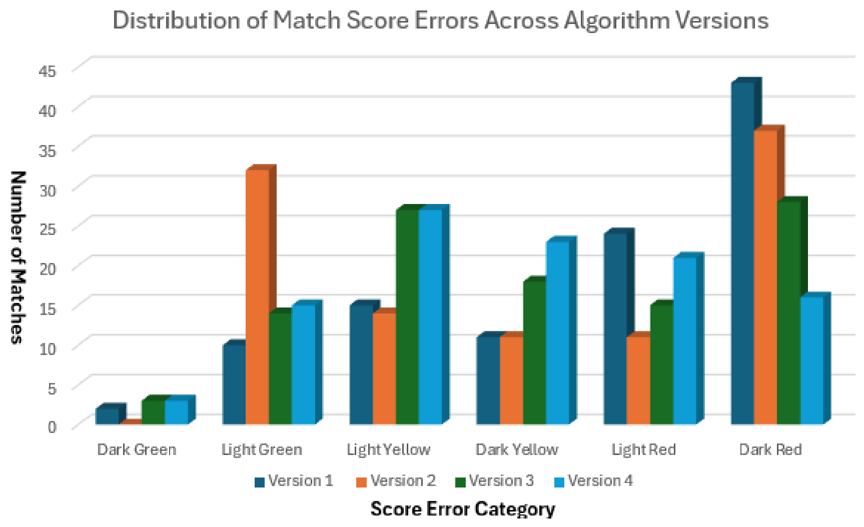
Skill C with a weight of 1

Then it will be computed as:

$W_{JD} = 5 + 4 + 1 = 10$ and the user has skill A and skill C but not skill B then $W_{Match} = 5 + 1 = 6$ he and the $Match\ Score = \left(\frac{6}{10}\right) \cdot 100\% = 60\%$ This example shows that missing very high weight such as skill B that has a weight of 4 would cause a significant drop in the match score, while missing a less important skill like skill C in this case will cause a less significant impact this can be seen in Fig 4. (a) with Table 1. Defining how the scoring parameters were implemented, this shows how the weighting system impacts score differentiation and improves match precision [27].

Table 1. Heatmap key that will be used in all future heatmaps.

Similarity Range	Colour
0%	Dark Green
±2-5%	Light Green
±6-10%	Light Yellow
±11-15%	Dark Yellow
±16-20%	Light Red
20%+	Dark Red



(a)

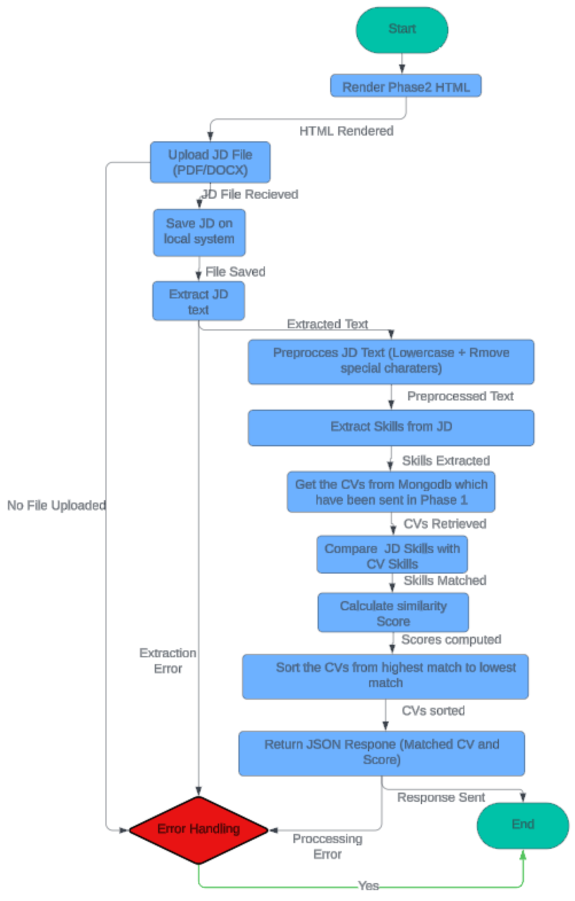
Fig 4. Mean skill Extraction accuracy of the algorithm based on version

Evaluation and Accuracy Improvements. To evaluate the effectiveness of the Phase 2 algorithm, a series of test JDs and synthetic CVs are used. Each test JD covers various roles and experience levels was run through the weighted matching system against a pool of candidate CVs. We examined whether the algorithm correctly assigns higher scores to the CVs that truly fulfil the important requirements of the JD. The results were visualized as heatmaps, where each cell represents the match score between a particular JD and a candidate CV. A colour gradient key in Table 1. As discussed earlier will help in interpreting these heatmaps, cells shaded towards the red end of the spectrum indicated high percentage mismatch while lighter coloured cells where they are closer to green indicate a similar or close match [28].

The pattern of results confirmed significant accuracy improvements with the weighted approach. In earlier unweighted matching. A candidate who listed many skills could score similarity to one who had fewer but very relevant skills, leading to potential false positives in ranking. After introducing skill weighting in Phase 2 final algorithm, the heatmaps showed a clearer contrast – candidates with the key skills required by the JD stood out with markedly higher scores, and those missing critical skills dropped lower in ranking. For instance, if a JD emphasized a particular high weight skill, any CVV that lacks that skill will show a substantial score reduction, correctly reflecting a weaker match. This improved mechanism makes it evident in the scoring logic updates are illustrated in Fig 4. Which were refined to balance the influence of critical vs minor skills.

We also refined the matching process through fuzzy matching thresholds and filtering to boost accuracy. The fuzzy matching threshold is calibrated where it would require a minimum similarity score for a fuzzy match to avoid spurious matches and to help in ensuring very similar skill names are counted. Additionally, a minimum match score cutoff can be applied as a filtering threshold – for example, recruiters might ignore candidates below say a 30% match and focus on strong matches. By tuning these parameters Phase 2 achieved a much more precise alignment between JD requirements and CV content [29].

Overall, the weighed skill matching approach in Phase 2 demonstrated a clear increase in matching accuracy compared to the baseline. The combination of an expanded skill dictionary, context aware extraction with fuzzy matching, and role specific weighting led to a higher recall of relevant skills and better precision in evaluating fit.



(a)

Fig 5. Phase 2 Final Flowchart

5.7 Phase 3: Transformer Based Semantic Similarity

Phase 3 implements a transformer-based approach to measure the semantic similarity between JDs and CVs, enabling the system to go beyond simple keyword matching. In this phase, textual content from each JD and CV is converted into a high dimensional embedding representation using a pretrained transformer model [10,30]. These embeddings capture the meaning and context of the documents, allowing recognition of similar skills and qualifications even when expressed in different terms [30-31]. By encoding both the CVs and JDs into shared vector space, the system can assess how a candidate's CV aligns with a JDs requirements based on semantic content rather than exact word overlap [32-33]. This provides a much more holistic matching of candidate qualifications to job needs, addressing the limitations of earlier keyword centric methods [24,27].

Embedding CVs and JDs with a Pretrained Transformer Model. At the core of Phase 3 is sentence BERT transformer model that generates embeddings for input text [30,33]. We utilize a pretrained model from the sentence transformer model library Hugging Face – initially using all-MiniLM-L6-v2 which is a MiniLM based sentence model for its efficiency and later adopting the higher capacity all-mpnet-base-v2 model for improved accuracy [10]. Each JD or CV, once procced into plain text will be passed into this model where to obtain a fixed size vector which is 768 dimensions in this model's case [30].

The transformer has been fine-tuned on massive datasets to produce semantically meaningful sentence embeddings, meaning that documents with similar content map to nearby points in the vector space [30,32]. In effect, the model encodes the context of skills, experience, and qualifications present in a CV or job posting into numerical form [10,33]. This approach has been shown to be effective for resume screening tasks – for example, prior work used Sentence-BERT embeddings to rank CVs by 768 dimensions similarity to a JD [10,26,30]. By using a pretrained model, our system leverages prior language understanding without needing to train a model from scratch. We chose the mpnet based model after comparative experiments indicated it yielded closer alignment to human relevance judgements than the smaller MiniLM model [31,33]. This could be since the computational load was justified by the gain in embedding quality and matching accuracy. In cases where scalability is critical, the MiniLM model remains a viable option, as it is optimized for efficiency in large scale matching tasks [10,32]. All embeddings are generated using the same to ensure that they reside in the same vector space making them directly compatible.

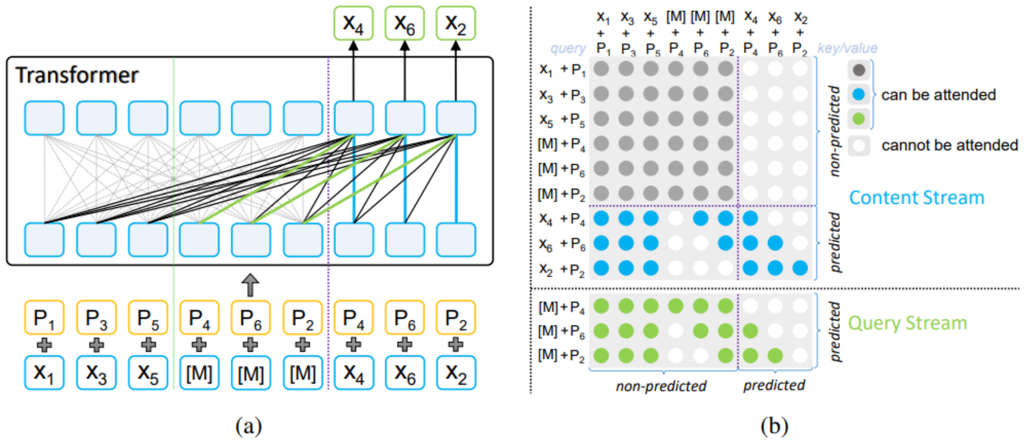


Fig 6. (a) The structure of MPNet (b) The attention mask of MPNet. [36]

MPNet Architecture Justification and Attention Mechanism Explanation. To improve semantic scoring in Phase 3, we selected the all-mpnet-base-v2 model due to its enhanced attention structure and pretraining objectives, which offer a accurate contextual embeddings than traditional BERT or other autoregressive models [31]. MPNet combines the strengths of masked language modelling (MLM) and permuted language modelling (PLM) by integrating a two-stream attention mechanism, allowing it to retain bidirectional context while simultaneously modelling dependencies for masked token prediction.

As visualized in Fig. 6 (a) and (b), MPNet introduces a dual stream attention mechanism where the content stream process the dull sequence, attending top all n on predicted tokens e.g. x_1, x_3, x_5 while the query stream is responsible for predicting the masked tokens e.g. x_4, x_6, x_2.In Fig. 6 (a), the grey lines illustrate bidirectional attention to over known content tokens, and in Fig. 6 (b), this is mirrored by the light grey attention m ask that allows full access within the content stream.

The blue and green lines in Fig. 6 (a) represent the content and query attention paths predicted tokens, and these are further explained by the corresponding masks in Fig. 6 (b). Notably, the query stream has restricted access – it cannot attend to other predicted tokens, maintaining casualty whereas the content stream can attend freely across all positions, enabling stronger global understanding [31]. The overlapping attention paths are shown in black, indicating shared attention flow between both streams.

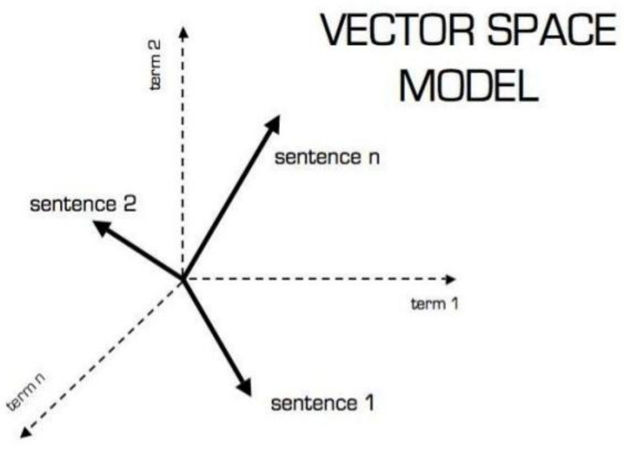
This structural design allows MPNet to outperform standard BERT like models by enabling richer representations for masked positions, especially in tasks involving semantic similarity such as CV and JD comparison [10,31]. In our implementation, this translates to embeddings that better capture both global document context and nuanced role specific skills [30-33]. Compared to earlier models like MiniLM or even RoBERTa, MPNet demonstrated a much more superior ability to align CV texts with JD requirements especially when job expectations were implied or expressed with varied phrasing [31].

Ultimately, this architectural advantage contributed very significantly to Phase 3 pipeline performance and interpretability, particularly in embedding alignment, which is quantitative reflected in Table 5 and visually validated through PCA in Fig. 13 (a) and (b) [32].

Text Preprocessing and Encoding Pipeline. Each CV and JD undergoes preprocessing before encoding to improve the quality the embeddings. First, the input files PDF and DOCX are parsed. With the raw text being cleaned and normalized and any excessive whitespace will be removed, with stop word removal that hold little to no semantic content [9,11].

Given that CVs and JDs can be multi sentence documents, we ensure the text is structured in a way that the model can handle [33]. The sentence Bert model can encode an entire paragraph or list of bullet points as a single sequence, up to limit of typically 512 tokens – effectively summarizing the content into one embedding [30]. In our implementation, we typically join the sentences of a CV or JD into one consolidated text input. If a document is very long, an optional strategy is to split it into smaller segments either by section or sentence and encode each segment, then combine the segment embeddings for instance by averaging to obtain an overall document vector [33]. This ensures we do not exceed the models token limit while preserving the semantic information of each part, in practice most of our CVs and JDs fall within the models input size, so a single pass encoding can be used for each. This would result in every JD and every CV being represented by a dense numeric embedding capturing its overall meaning and key topics [33].

Semantic Similarity Calculation (Cosine Similarity). Phase 3 also uses an embedding to compute it into the cosine similarity between the JD and CV vectors to quantify their semantic similarity as is explained in equation (1). The normalised dot product of the two vectors scores ranges from -1 to 1 where 1 indicates identical direction and 0 indicates no significant similarity [32-33]. In our context, because CV and JD text are expected to be at least somewhat related, the similarity scores generally fall between 0 which means there is a very low match and 1 a very high match. A higher cosine value would imply that the CV embedding points in a similar direction to the JD embedding, meaning the candidates qualifications closely match the JD requirement in meaning. Cosine similarity is computationally efficient and works well for semantic embeddings, as it focuses on the orientation of the vectors rather than their magnitude [32]. We pytorch implementation from the sentence transformers utility library to calculate these similarity scores for each JD and CV pair. The concept of cosine similarity is illustrated here in the vector space model as shown in Fig 7. (a), where different sentences are represented as directional vectors in high dimensional space. In this diagram, the smaller the angle between the two vectors like in sentence 1 and in sentence 2, the higher their semantic similarity. This figure illustrates how CV and JD embeddings that point in similar directions in vector space are interpreted by the system as strong matches [32].



(a)

Fig 7. Illustration of vector space model [35]

Ranking of Candidates by Similarity Score. The resulting similarity score is used to candidates for each JD. For each JD in the input set, the system compares its embedding to the embedding of every CV, compute the cosine similarity, and then sorts the CVs in descending order of similarity [32-33]. A resume with s=0.85, for example, would be considered a closer match to the JD requirements that one with s=0.6. The output of Phase 3 is a list of CVs for each JD, ordered from most to least similar, along with similarity scores. This ranked list directly addresses the recruitment problem of finding the best matching applicants: the recruiter can focus attention on the top ranked CVs, which the model has identified as the most contextually relevant to the position [26,29]. In the backend implementation, this was achieved by iterating over each JD embedding and computing its cosine similarity with each CV embedding in turn, then collecting these scores. The system presents the scores in a tabular format via the front-end interface for user interpretation [11]. Importantly, these similarity scores are relative indicators of fit. We refrain from assigning any absolute match/pass marking in this phase instead, Phase 3 provides a continuous metric that can be used in decision making [33]. In practice, one might apply a cutoff threshold on the score to filter out very low matches – for example ignoring CVs below a similarity of 0.3 or 0.4 if they are seen as being unrelated. In this design the threshold can be adjusted based on domain needs, but by default we present all scored results and allow the natural ranking to highlight the best candidates [27].

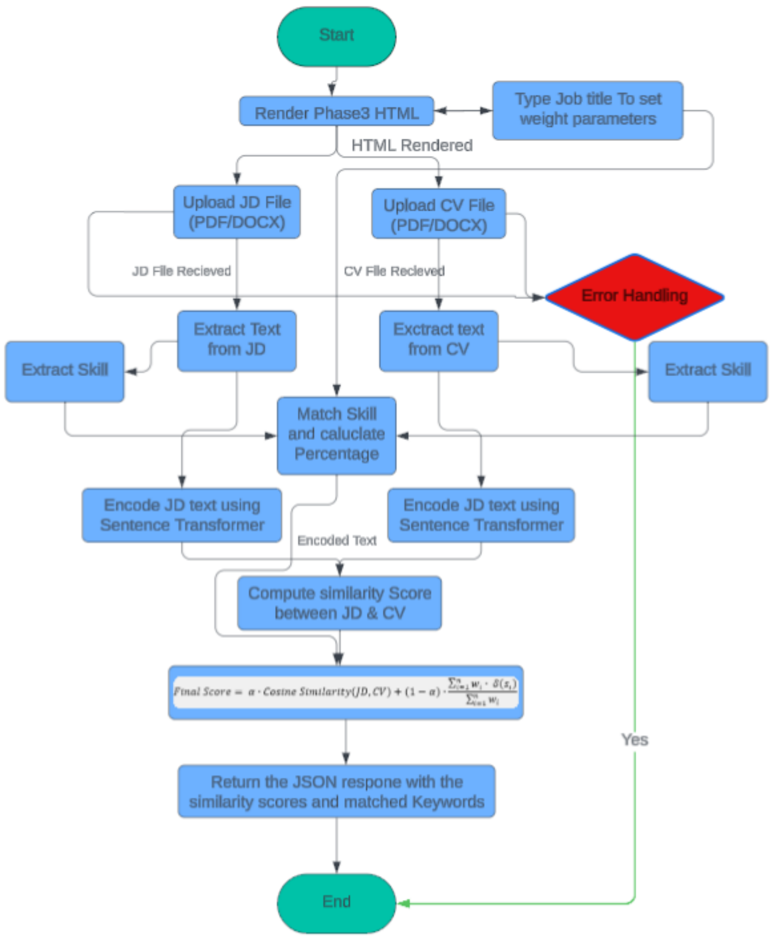
Implementation Decisions and Accuracy Enhancements. Several implementation decisions were made in Phase 3 to improve matching accuracy and the systems performance [30-31,33]. One of the major decisions done was the choice of the embedding model. We initially started with a lighter sentence-Bert model that was just used for development later switched to a more powerful model after evaluating multiple transformers on a validation test as shown in Table 1. The reason I chose all-mpnet-base-vb2 over paraphrase-mpnet-base-v2 even though it gave slightly better results is because of it being trained on much more diversified tasks, its ability to handle long documents, better generalization and superior performance in benchmarks. The tests shown in Table 1. Indicates the all-mpnet-base-v2 model is a close second place and factoring all its advantages over the first-place model it makes more sense long term to choose it for the final pipeline. Additionally, we considered whether to fine tune the transformer on a domain specific dataset of CVs and JDs. Fine tuning can further improve accuracy by adapting the embeddings to the nuances of a specific industry [33,34].

To further improve matching precision, Phase 3 incorporates a hybrid similarity scoring mechanism that combines the transformer based semantic similarity with keyword-based skill matching [9-11]. Specifically, we integrate the skill extraction results from Phase 2 into the final scoring formula [22,24]. The system first identifies key skills mentioned in the JD using the Phase 2 methodology. It then checks which of these skills appear in the candidates CV [23]. We then compute the skill overlap score –essentially the fraction of important JD skills presents in the CV [24,28]. This overlap is used to adjust the raw cosine similarity. In the implementation, we assigned a weight to the semantic similarity component and a weight to the skill overlap component, combing them to produce a final composite score for each CV to JD pair [26,33]. This hybrid approach proved to be effective in improving the alignment on niche skills, and it ensures that critical qualifications are not overlooked by the general semantic model [30,32]. The introduction of skill overlap scoring in Phase 3 which is denoted in version 2 of the Phase 3 model in our development logs showed a notable increase in matching accuracy in testing. We iteratively refined the weight parameters for this combination over several versions to help in maximizing the agreement with manual scoring that was done for each CV to JD pair with a marked criterion as seen in Table 3. The final system thus produces a similarity score that reflects both contextual similarity and specific skill matching.

Table 2. Quantitative comparison of raw sentence embedding models based on Heatmaps using **Table 1.** Key.

	Dark Green	Light Green	Light Yellow	Dark Yellow	Light Red	Dark Red
paraphrase-MiniLM-L6-v2	1	11	11	**17**	14	51
all-roberta-large-v1	0	6	10	8	14	**67**
paraphrase-mpnet-base-v2	**2**	**17**	**19**	6	14	47
multi-qa-mpnet-base-dot-v1	1	9	8	16	6	65
all-mpnet-base-v2	**2**	10	15	11	**24**	43

Finally, we set up the Phase 3 backend as a Flask web service integrated with the front-end interface developed Phase 3 to handle user request [11]. When a user submits JDs and CVs for analysis, the backend executes the pipeline: text extraction, preprocessing, embedding via the transformer model, cosine similarity computation, skill matching, and result aggregation [10,33]. The design ensures that all these steps occur seamlessly for the user, delivering a ranked list of candidates for each JD with an explanation score. All numerical similarity calculations and ranking logic are confined to the backend, consistent with the design that separate the NLP processing form the user interface. In summary, Phase 3 methodology harness a state-of-the-art transformer model for semantic textual comparison and augments it with domain specific keyword matching. This yields a powerful semantic similarity engine that forms the backbone of our CV ranking system, enabling more accurate and intelligent matching of candidates to JD without relying simply on keyword hits [27]. The approach is formalized and modular, allowing further tuning such as adjusting similarity thresholds or retaining embeddings on new data to continually improve the matching performance as needed [34].


Start
Render Phase3 HTML
Type Job title To set weight parameters
HTML Rendered
Upload JD File (PDF/DOCX)
Upload CV File (PDF/DOCX)
JD File Recieved
CV File Recieved
Error Handling
Extract Text from JD
Exctract text from CV
Extract Skill
Extract Skill
Match Skill and caluclate Percentage
Encode JD text using Sentence Transformer
Encode JD text using Sentence Transformer
Encoded Text
Compute similarity Score between JD & CV
$Final\ Score = \alpha \cdot Cosine\ Similarity(JD, CV) + (1 - \alpha) \cdot \frac{\sum_{i=1}^{n} w_i \cdot \delta(s_i)}{\sum_{i=1}^{n} w_i}$
Yes
Return the JSON respone with the similarity scores and matched Keywords
End

(a)

Fig 8. Phase 3 Final Flowchart

6 Results and Discussion

Phase 1 Skill extraction overall for each version. Fig 9. (a)-(d) demonstrates the progressive improvement in skill extraction accuracy across four system versions. Version 1 in Fig. 9 (a) served as the baseline for this system where it used a small manually created list with no fuzzy logic, which resulted in poor skill extraction accuracy ranging from 6% to 33%, particularly underperforming for niche roles like cybersecurity and cloud architect. Version 2 Fig. 9 (b) introduced fuzzy matching but lacked threshold tuning and a large dictionary coverage, which yielded only marginal or even reduced gains, with accuracy mostly between 10% and 29%. A significant boost occurred in Version 3 Fig. 9 (c), where a expanded predefined skill list, improved fuzzy logic using **token_set_ratio** [14], and contextual checks raised accuracy to between 28% and 74%, especially benefiting previously weak roles. Finally, Version 4 Fig. 9 (d) achieved consistent and robust extraction across all CVs, with accuracy overall surpassing 70% and reaching up to 88%, meeting the projects target of 70% through refined thresholds and a fully optimized skill list.

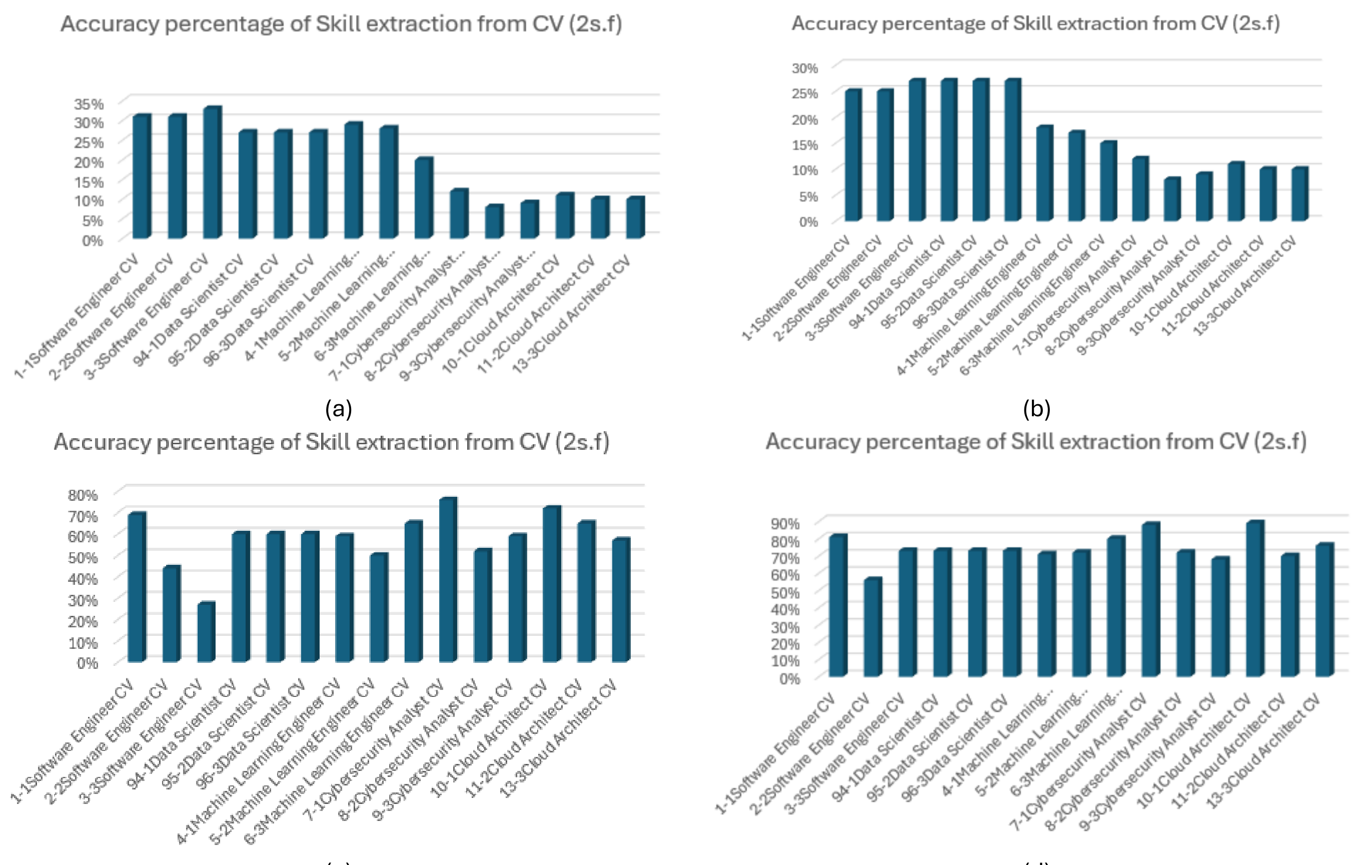


Fig 9. The accuracy percentage of skill extraction from each CV, calculated as the ratio of correctly identified skills to total skills in the CV

Phase 1 Skill extraction for each version with key details. Fig. 10 (a)-(d) presents a much more comprehensive skill matching evaluation for the performance of four versions of the Phase 1 algorithm, detailing how well the system identified and correct skills from CVs. In Version 1 Fig. 10 (a), the algorithm showed a high rate of missing technical skills, and a low count of correct skills caught, with a few wrongly caught skills which shows its limited detection capacity and vocabulary scope. Version 2 Fig. 10 (b) improved marginally by slightly increasing the number of correctly matched skills but still suffered from high miss rates especially in complex roles like cybersecurity analyst. Version 3 Fig. 10 (c), showed a strong shift where the algorithm caught significantly more valid skills, reduced missed skills and maintained a much more manageable level of false positives thanks to the enhanced fuzzy matching logic and larger predefined skill list. Version 4 Fig. 10 (d) marked a peak in performance, achieving high recall with most CVs showing very low missing skills, minimal false positives, and a high count of correctly matched skills that often approached or matched the actual skill count per CV.

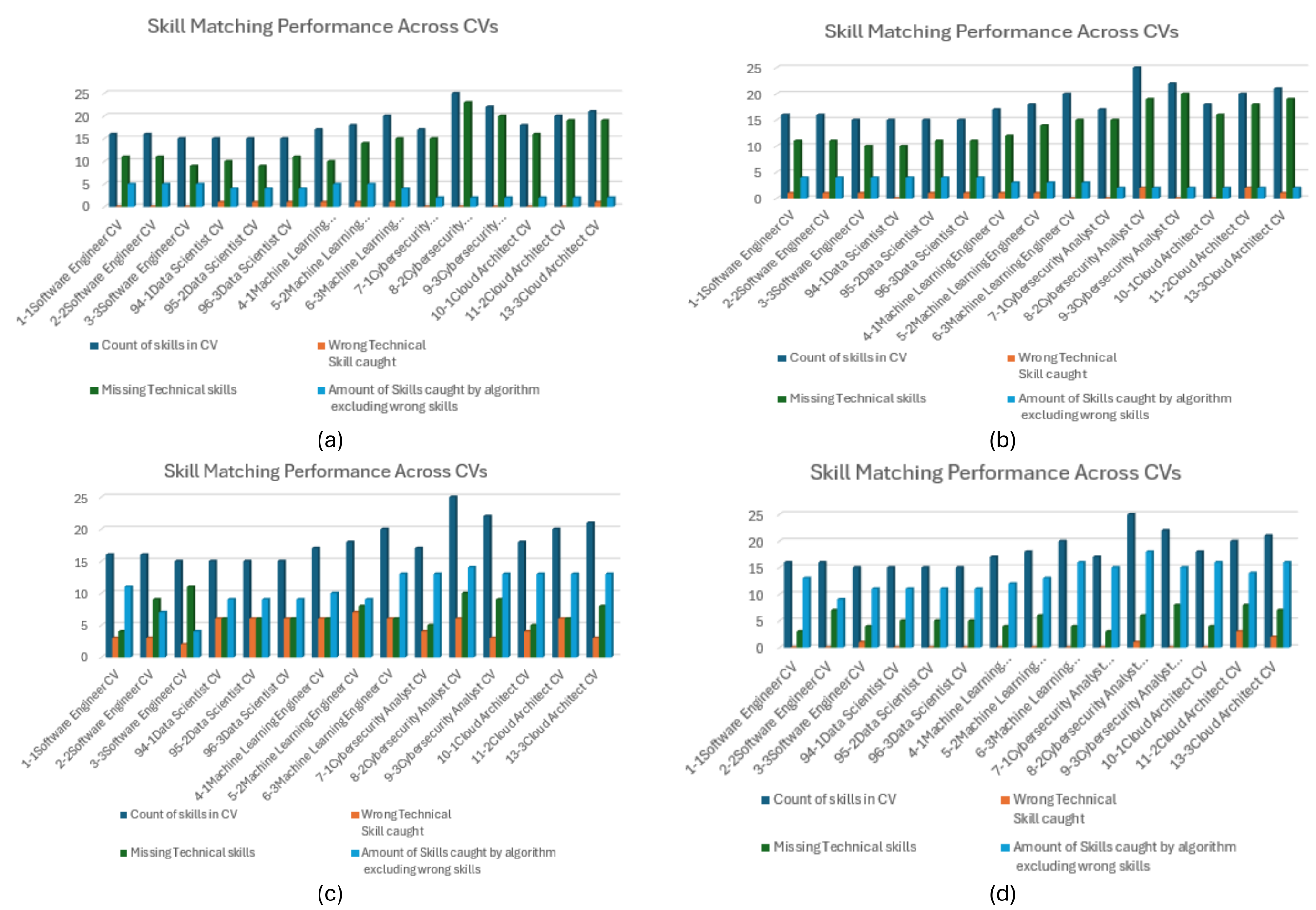


Fig 10. Skill matching performance across CVs comparing total skills in the CV with the correctly matched skills, wrongly matched skills and missing skills.

Table 3. Benchmark Reference scores

	Software Engineer JD	Data Scientist JD	Machine Learning JD	Cybersec-urity Analyst JD	Cloud Architec t JD	AI Engineer JD	Data Engineer JD
1 Software Engineer CV	43	90	76	17	23	56	46
2 Software Engineer CV	43	90	76	17	23	56	46
3 Software Engineer CV	43	90	76	17	23	56	46
1 Data Scientist CV	43	90	86	17	30	63	63
2 Data Scientist CV	43	90	86	17	30	63	63
3 Data Scientist CV	43	90	86	17	30	63	63
1 Machine Learning Engineer CV	50	83	76	17	30	63	36
2 Machine Learning Engineer CV	50	83	76	17	30	63	36
3 Machine Learning Engineer CV	50	83	76	17	30	63	36
1 Cybersecurity Analyst CV	30	17	17	83	10	17	30
2 Cybersecurity Analyst CV	30	17	17	83	10	17	30
3 Cybersecurity Analyst CV	30	17	17	83	10	17	30
1 Cloud Architect CV	30	17	17	79	10	17	36
2 Cloud Architect CV	30	17	17	79	10	17	36
3 Cloud Architect CV	30	17	17	79	10	17	36

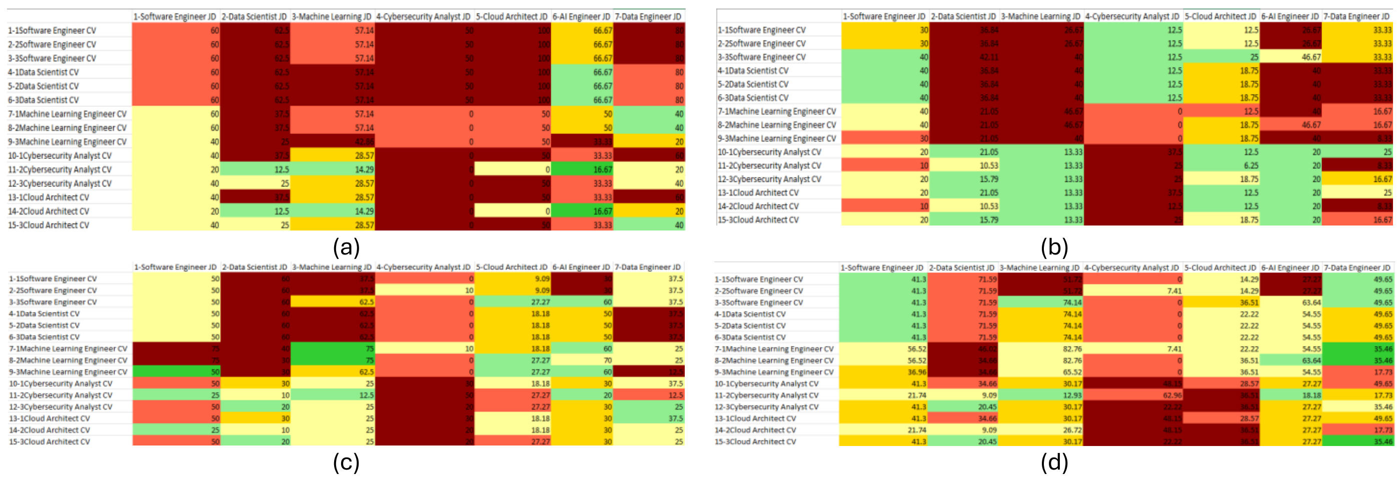


Fig 11. Heatmap for Phase 2 (a) Version 1, (b) Version 2, (c) Version 3, (d) Version 4.

Phase 2 JD matching for each version with key details. Fig 11. (a)-(d) presents a much more detailed evaluation of semantic similarity performance for four versions of Phase 2 algorithm, which compares CVs against JDs to determine alignment. In Version 1 Fig. 11 (a), the system generated inflated similarity scores for some matching roles such as software engineering but lacked precision across most other roles, often assigning overly high or flat scores even when the CV and JD were not relevant, showing limited contextual differentiation and poor weighting logic. Version Fig. 11 (b) improved only marginally by introducing slight threshold tuning, but still struggled with highly inaccurate matches, especially for niche roles like cybersecurity analyst, where scores remained low or erratic due to insufficient role specific tuning. Version 3 Fig. 11 (c) showed a marked improvement, introducing fuzzy matching and preliminary context filtering that resulted in more accurate score distribution, improved cross role separation, and reduced false similarity scores, especially between unrelated job types. Version 4 Fig.11 (d) marked the best performance of Phase 2, incorporating job specific skill weights and normalization, which led to highly realistic scores across matching roles, with CVs scoring highest against their corresponding JD types, closely resembling the benchmark reference patterns as seen in Table 3.

Table 4. Quantitative comparison of raw sentence embedding models based on Heatmaps using **Table 1.** Key. For Phase 3

	Dark Green	Light Green	Light Yellow	Dark Yellow	Light Red	Dark Red
Version 1	2	10	15	11	**21**	**46**
Version 2	0	**32**	14	11	11	37
Version 3	**3**	14	**27**	18	15	28
Version 4	**3**	15	**27**	**23**	**21**	16

Phase 3 semantic similarity scoring for each version with key details. Fig 12. (a)-(d) presents a comprehensive comparison of the semantic matching accuracy between CVs and JDs across four versions of the Phase 3 algorithm, which leverages sentence embeddings and advanced logic scoring logic. In Version 1 Fig. 12 (a), the model used basic MPNet sentence embeddings with cosine similarity, which resulted in some reasonably aligned scores for clearly matching roles like Software Engineer or Data Scientist but overall showed uneven performance and a tendency to over score unrelated roles like cybersecurity or cloud architect, indicating the need for more nuanced scoring and domain adaptation. Version 2 Fig. 12 (b) incorporated a hybrid scoring system that combined semantic similarity with extracted skill match score, producing more stable and role aligned results. However, certain matches still appeared inflated or indistinct, particularly in roles where soft skills dominate or where technical overlap exists. Version 3 Fig. 12 (c) introduced job specific weighted scoring and further tuning, which led to clear improvements in match precision, as seen in stronger diagonal clustering and more realistic separation of unrelated roles, reducing false positives significantly. Finally, Version 4 Fig. 12 (d) achieved the highest alignment with reference expectations, showing distinct and consistently strong semantic matches between corresponding CVs and JDs, minimal noise across unrelated domains, and refined scoring distributions. The improvements were especially clear for complex or hybrid roles like Cloud Architect and Ai Engineer, marking this version as the most effective implementation of Phase 3s semantic similarity scoring system.

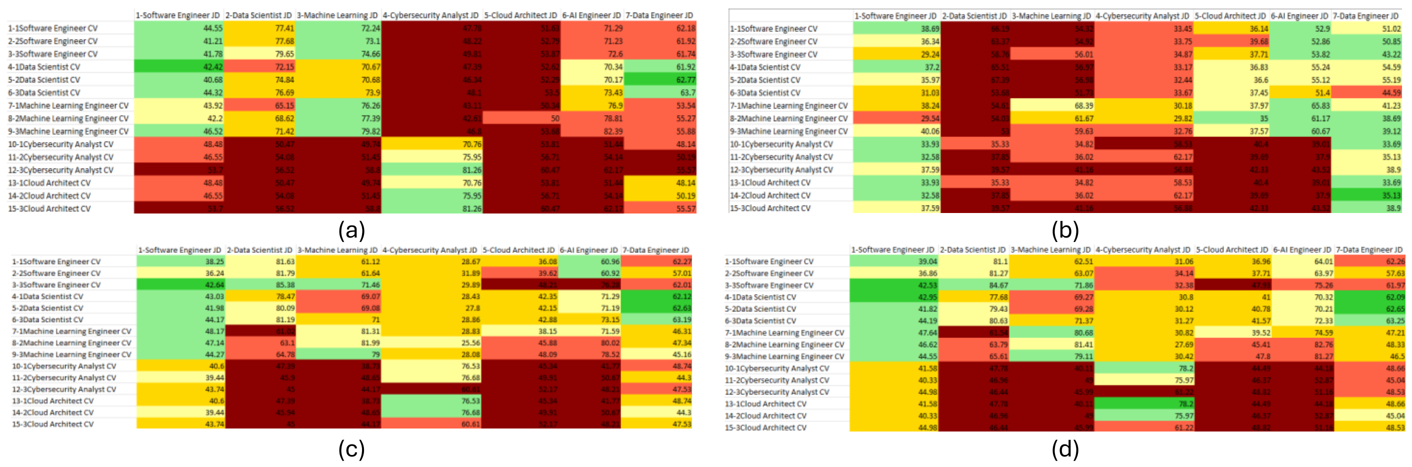


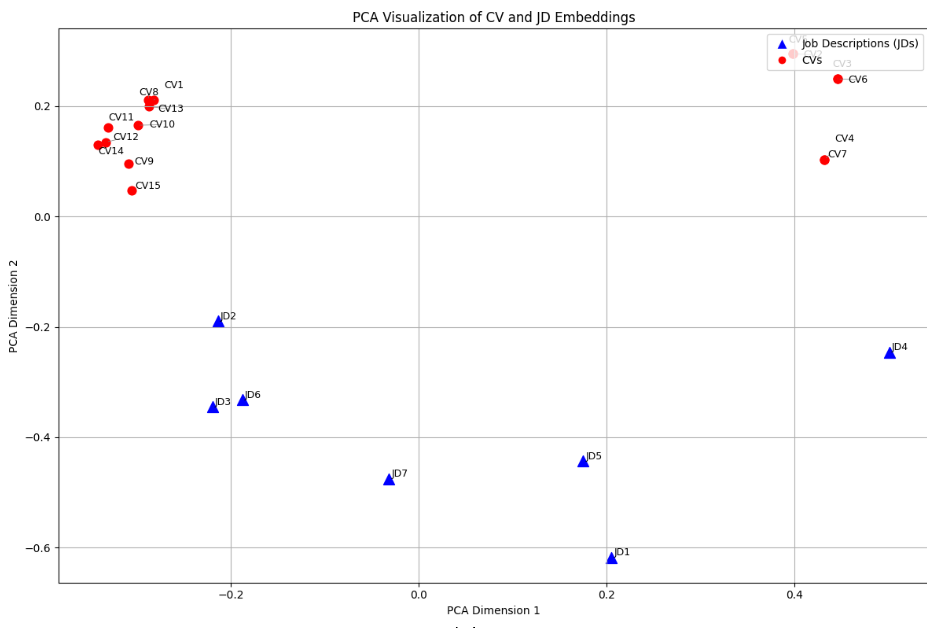
Fig 12. Heatmap for Phase 3 (a) Version 1, (b) Version 2, (c) Version 3, (d) Version 4.

Table 5. Quantitative comparison of raw sentence embedding models based on Heatmaps using **Table 1.** Key. For Phase 3

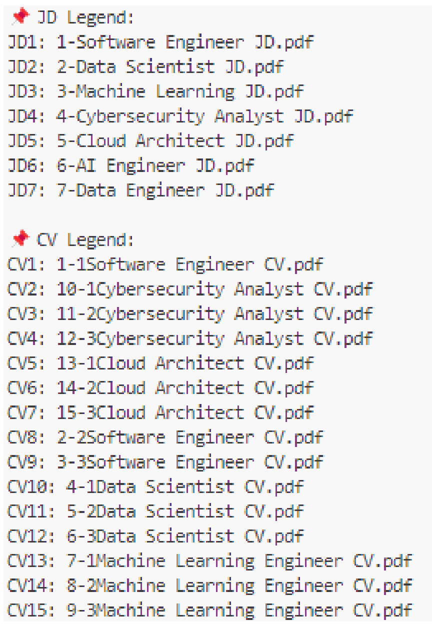
	Dark Green	Light Green	Light Yellow	Dark Yellow	Light Red	Dark Red
Version 1	2	17	7	14	20	**46**
Version 2	1	**20**	**19**	9	**21**	35
Version 3	3	14	18	27	14	29
Version 4	**5**	13	13	**29**	17	28

PCA based embedding visualization. To further interpret the behaviour of the transformer embedding and validate semantic alignment, we visualized the high dimensional CV and JD vectors using principal component analysis. Each embedding, originally a 768-dimensional vector, was projected into two dimensions to observe spatial relationships between CVs and JDs in the semantic space. As shown in Fig 13. (a), the red circles represent the CVs, and the blue triangles represent the JDs. Each point is labelled with a simple identifier like CV1, CV2, JD1 and more corresponding legend in Fig 13. (b) is provided to map each ID to its full filename.

The results support the systems accuracy for example we can see that for JD2 the data scientist JD that it appears closest to multiple data scientist CVs such as CV4, CV6, CV14 that have relevance to that field which indicates that there is a strong semantic similarity between them. This visualization provides an intuitive and interpretable method to verify that the model maps related documents into nearby regions of embedding space – validating both the semantic encoders effectiveness and the match score’s reliability in Phase 3.



(a)



```
📌 JD Legend:
JD1: 1-Software Engineer JD.pdf
JD2: 2-Data Scientist JD.pdf
JD3: 3-Machine Learning JD.pdf
JD4: 4-Cybersecurity Analyst JD.pdf
JD5: 5-Cloud Architect JD.pdf
JD6: 6-AI Engineer JD.pdf
JD7: 7-Data Engineer JD.pdf

📌 CV Legend:
CV1: 1-1Software Engineer CV.pdf
CV2: 10-1Cybersecurity Analyst CV.pdf
CV3: 11-2Cybersecurity Analyst CV.pdf
CV4: 12-3Cybersecurity Analyst CV.pdf
CV5: 13-1Cloud Architect CV.pdf
CV6: 14-2Cloud Architect CV.pdf
CV7: 15-3Cloud Architect CV.pdf
CV8: 2-2Software Engineer CV.pdf
CV9: 3-3Software Engineer CV.pdf
CV10: 4-1Data Scientist CV.pdf
CV11: 5-2Data Scientist CV.pdf
CV12: 6-3Data Scientist CV.pdf
CV13: 7-1Machine Learning Engineer CV.pdf
CV14: 8-2Machine Learning Engineer CV.pdf
CV15: 9-3Machine Learning Engineer CV.pdf
```

(b)

Fig 13. PCA Visualization of Semantic Embeddings for CVs and JDs Using all-mpnet-base-v2

7 Conclusion and Recommendations for Further Work

7.1 Conclusions

The development of the AI driven curriculum vitae analysis model has successfully addressed the critical challenges of unstructured document parsing, skill extraction, and sematic job matching across three incremental phases. From initial fuzzy logic-based skill identification in synthetic CVs, to weighted job skills comparisons, and finally transformer powered semantic ranking, each phase contributed significantly to the overall adaptability and interpretability of the system. By employing tools such as OCR, PyMuPDF and docx and more for enhanced preprocessing, the ESCO API, MongoDB storage, and advanced embedding models like **all-mpnet-base-v2**, the project was able to convert unstructured CV documents into structured, analysable data with measurable accuracy and consistency.

Each phase demonstrated progressive improvements, not only in technical performance but also design modularity and user experience. Phase 1 reached skill extraction accuracy 74% through optimization being doing iteratively. Phase 2 introduced weight scoring that increased the realistic nature of CV JD matching. Phase 3 solidified semantic understanding combining dense vector similarity and weighted skill overlap to provide better ranking. The integration of predefined and fetched job skills, role specific weight systems, and scalable database logic ensures that the platform is not only functional but also extensible to the real world for recruitment platforms. Additionally, the systems use of synthetic data and SHA-256 hashing for deduplication ensured compliance with GDPR requirements.

This work demonstrates a proof-of-concept platform that bridges the gap between human level contextual assessment and scalable automation. This project validates the fact that AI can be employed not just to rank CVs but to provide fair and unbiased assessments that are more aligned with modern hiring expectations.

7.2 Recommendations for Further Work.

Model Fine Tuning: While **all-mpnet-base-v2** offered strong results, domain specific fine tuning on recruitment datasets could enhance semantic alignment. Training on CV JD pairs labelled with match scores could significantly improve embedding performance and reduce false matches.

XAI: integrating explainable AI methods, such as skill level contribution charts or sentence level highlighting, would help users understand why a particular CV ranks highly, fostering trust and transparency.

Cross Language Support: Extending the system to support multilingual CVs and JDs using language detection and multilingual transformer models.

Real World Data Testing: The current system relies on synthetic CVs and JDs, Future work should focus on validating the pipeline using anonymised or consented real world documents to help in assessing new formats, vocabulary, and role ambiguities.

Skill clustering and career path suggestions: By clustering extracted skills the system can provide candidates with feedback or suggest career transitions based on skill overlap with other roles.

Table 6. Quantitative Summary Table for Each Phase

Phase	Version	Key Features Added	Accuracy/Metric
Phase 1	V1	Predefined list of skills + ESCO API skills	~21%
Phase 1	V2	simple fuzzy matching mechanism	~19%
Phase 1	V3	Advanced fuzzy + length penalty + regex	~58%
Phase 1	**V4**	**Expanded and fine-tuned predefined skill list**	**74%**
Phase 2	V1	Small skill list (13)	Low heatmap density
Phase 2	V2	Larger skill list, no fuzzy	Moderate improvement
Phase 2	V3	Fuzzy + regex matching	Better Match density
Phase 2	**V4**	**Skill weighting added**	**Significantly better matching**
Phase 3	V1	Cosine similarity (all-mpnet-base-v2 only)	Baseline semantic similarity
Phase 3	V2	Skill match combined with decent skill list, α=0.7 for model and α=0.3 for skills	More interpretable
Phase 3	V3	Improved predefined skill list, Skill weighting final score α=0.6 for model and α=0.4 for skills	Higher fidelity matches
Phase 3	**V4**	**Manual tuning to weighting system to α=0.65 for model and α=0.35 for skills**	**Best Correlation to manual scoring in Table 3. Overall.**

References

[1] "AI in Talent Acquisition: Top Challenges for 2025," Korn Ferry, Dec. 9, 2024. [Online]. Available: https://www.kornferry.com/insights/featured-topics/talent-recruitment/ai-in-talent-acquisition-top-challenges-for-2025. [Accessed: Apr. 2, 2025].

[2] "AI in Recruiting 2024: Pros and Cons," Korn Ferry, Mar. 2024. [Online]. Available: https://www.kornferry.com/insights/featured-topics/talent-recruitment/ai-in-recruiting-navigating-trends-for-2024. [Accessed: Apr. 2, 2025].

[3] "5 challenges of AI in recruitment," TechTarget, Mar. 2025. [Online]. Available: https://www.techtarget.com/searchhrsoftware/feature/Challenges-of-AI-in-recruitment. [Accessed: Apr. 2, 2025].

[4] "The Benefits and Disadvantages of Applicant Tracking Systems," 4 Corner Resources, Aug. 2023. [Online]. Available: https://www.4cornerresources.com/blog/the-pros-and-cons-of-applicant-tracking-systems/. [Accessed: Apr. 2, 2025].

[5] "The Problem with Most Applicant Tracking Tools (ATS)," Eternal Works. [Online]. Available: https://www.eternalworks.com/blog/the-problem-with-most-applicant-tracking-tools-ats. [Accessed: Apr. 2, 2025].

[6] "AI in recruitment: navigating the advantages and challenges," Deeper Signals. [Online]. Available: https://www.deepersignals.com/blog/ai-recruitment-advantages-challenges. [Accessed: Apr. 2, 2025].

[7] "AI for Recruiting: A Definitive Guide to Talent Acquisition in 2025," Vonage, Feb. 2025. [Online]. Available: https://www.vonage.com/resources/articles/ai-for-recruiting/. [Accessed: Apr. 2, 2025].

[8] "The Limitations of Legacy ATS in Modern Talent Acquisition," Talview, Jan. 2024. [Online]. Available: https://blog.talview.com/en/limitations-of-legacy-ats-in-modern-talent-acquisition. [Accessed: Apr. 2, 2025].

[9] Lane, H., Howard, C., & Hapke, H. (2019). *Natural Language Processing in Action*. Manning Publications.

[10] Hugging Face. "all-mpnet-base-v2." [Online]. Available: https://huggingface.co/sentence-transformers/all-mpnet-base-v2

[11] Lane, H., Howard, C., & Hapke, H. (2019). *Natural Language Processing in Action*. Manning Publications, pp. 81-83, 443-445

[12] "PyMuPDF Documentation," [Online]. Available: https://pymupdf.readthedocs.io/en/latest/

[13] Python Software Foundation. "pytesseract Documentation." [Online]. Available: https://pypi.org/project/pytesseract/

[14] Seatgeek Inc. "fuzzywuzzy Python Library." [Online]. Available: https://github.com/seatgeek/fuzzywuzzy [Accessed: Apr. 3, 2025].

[15] Ratcliff, J. W., & Metzener, D. E. (1988). *Pattern-Matching: The Gestalt Approach*.

[16] European Commission. "ESCO API Documentation." [Online]. Available: https://ec.europa.eu/esco/portal

[17] MongoDB Inc. “Data Modeling in MongoDB: Best Practices.” [Online]. Available: https://www.mongodb.com/docs/manual/data-modeling-introduction/

[18] "The Limitations of Legacy ATS in Modern Talent Acquisition," Talview, Jan. 2024. [Online]. Available: https://blog.talview.com/en/limitations-of-legacy-ats-in-modern-talent-acquisition

[19] Deeper Signals. "AI in Recruitment: Advantages and Challenges." [Online]. Available: https://www.deepersignals.com/blog/ai-recruitment-advantages-challenges

[20] Alooba. "Fuzzy Matching: Understanding the Concept and its Application in Natural Language Processing." [Online]. Available: https://www.alooba.com/skills/concepts/natural-language-processing/fuzzy-matching/ [Accessed: Apr. 10, 2025].

[21] Recrew.ai. "Overcome Top Resume Parsing Challenges: Proven Strategies." [Online]. Available: https://www.recrew.ai/blog/top-7-resume-parsing-challenges-and-strategies-to-overcome [Accessed: Apr. 10, 2025].

[22] Harbinger Group. "Upgrade Resume Parsing: NLP-Based Skill Extraction." [Online]. Available: https://www.harbingergroup.com/blogs/replace-your-existing-resume-parser-with-intelligent-skill-extraction-engine-based-on-nlp/ [Accessed: Apr. 10, 2025].

[23] CodersRank, “Skills-Based Matching: The Smartest Way to Match Developers to Jobs,” *CodersRank*, [Online]. Available: https://blog.codersrank.io/skills-based-matching/. [Accessed: Apr. 10, 2025].

[24] TechTarget, “How AI can help match the right person to the right job,” *SearchHRSoftware*, Feb. 2024. [Online]. Available: https://www.techtarget.com/searchhrsoftware/feature/How-AI-can-help-match-the-right-person-to-the-right-job [Accessed: Apr. 10, 2025].

[25] Daxtra Technologies, “What is Resume Parsing and How Does It Work?,” *Daxtra Blog*, [Online]. Available: https://www.daxtra.com/blog/what-is-resume-parsing-and-how-does-it-work/ . [Accessed: Apr. 10, 2025].

[26] Lever, “The Science Behind Skills-Based Hiring,” *Lever Talent Blog*, [Online]. Available: https://www.lever.co/blog/the-science-behind-skills-based-hiring/ . [Accessed: Apr. 10, 2025].

[27] CEIPAL, “AI-Driven Resume Ranking: How It Works and Why It’s Better,” *CEIPAL Blog*, [Online]. Available: https://www.ceipal.com/blog/ai-driven-resume-ranking/ . [Accessed: Apr. 10, 2025].

[28] HRForecast, “AI-based skill matching: How it works,” *HRForecast*, [Online]. Available: https://hrforecast.com/ai-based-skill-matching/ . [Accessed: Apr. 10, 2025].

[29] TalentLyft, “How to Match Candidates with the Right Job Openings,” *TalentLyft Blog*, [Online]. Available: https://www.talentlyft.com/en/blog/article/297/how-to-match-candidates-with-the-right-job-openings . [Accessed: Apr. 10, 2025].

[30] S. Reimers and I. Gurevych, “Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks,” *arXiv preprint arXiv:1908.10084*, Aug. 2019. [Online]. Available: https://arxiv.org/abs/1908.10084. [Accessed: Apr. 14, 2025].

[31] K. Song et al., “MPNet: Masked and Permuted Pre-training for Language Understanding,” *arXiv preprint arXiv:2004.09297*, Apr. 2020. [Online]. Available: https://arxiv.org/abs/2004.09297. [Accessed: Apr. 14, 2025].

[32] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*, Cambridge University Press, 2008, pp. 117–120.

[33] SentenceTransformers Documentation, “Semantic Textual Similarity with Cosine Similarity,” *SentenceTransformers*, [Online]. Available: https://www.sbert.net/examples/applications/semantic-search/README.html [Accessed: Apr. 14, 2025].

[34] B. Wang, Y. Liu, and P. Zhang, “Fine-Tuning Pretrained Language Models: Benefits, Challenges, and Strategies,” *Journal of Artificial Intelligence Research*, vol. 70, pp. 495–520, 2021. [Online]. Available: https://jair.org/index.php/jair/article/view/12258 . [Accessed: Apr. 14, 2025].

[35] C. S. Perone, “Machine Learning: Cosine Similarity for Vector Space Models (Part III),” *christianperone.com*, Sep. 15, 2013. [Online]. Available: https://blog.christianperone.com/2013/09/machine-learning-cosine-similarity-for-vector-space-models-part-iii/ [Accessed: Apr. 14, 2025].

[36] K. Song, X. Tan, T. Qin, J. Lu, and T.-Y. Liu, "MPNet: Masked and Permuted Pre-training for Language Understanding," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 16857–16867. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/c3a690be93aa602ee2dc0ccab5b7b67e-Paper.pdf [Accessed: Apr. 14, 2025

Appendix A – Source Code for Backend and Frontend

Appendix A.1 – Phase 1: Skill Extraction Logic Backend

```
#This code Should work with the "LSBUPhase1.html template they everything worked last check was on 27/01/2025"
import re # For regular expressions used in skill extraction and keyword pattern matching
import fitz  # PyMuPDF for PDF handling
import docx  # for handling DOCX files
import os # For file path handling, enviroment variables and working directory checks
from PIL import Image # To handle image formats if OCR is needed
from PyQt5.QtWidgets  # PyQt5 import
from fuzzywuzzy import fuzz, process # Import fuzzy matching functions from the fuzzywuzzy library
from pymongo import MongoClient # To connect to MongoDB for storing CVs
from flask import Flask, render_template, request, jsonify, send_file, abort, url_for, redirect
# Flask framework used for setting up backend API
import hashlib # For hasjhing CV file content
import smtplib #  For sending emails
from email.mime.multipart import MIMEMultipart # For building multipart email
from email.mime.text import MIMEText # to define the body of the email
import schedule # For setting scheduled tasks
import time # used with scheduling and timing events
import requests  # Added for ESCO API integration
print("Current Working Directory:", os.getcwd())

# Global variable to store dynamically generated skills from ESCO API
extracted_skills_ESCO = []

app = Flask(__name__)

# MongoDB setup
client = MongoClient('mongodb://localhost:27017/')
db = client['cv_analysis2_db']  # Database name
cv_collection = db['cvs2']  # Collection name

# Directory to save uploaded files
UPLOAD_FOLDER = os.path.join(os.getcwd(), 'uploaded_files')
os.makedirs(UPLOAD_FOLDER, exist_ok=True)  # Ensure the folder exists

# Preprocessing: clean the text (lowercase, remove special characters)
def preprocess_text(text):
   text = re.sub(r'[^a-zA-Z0-9\s]', '', text)
   return text.lower()

# Function to save CV data into MongoDB in an organized way
def save_cv_to_mongodb(filename, structured_data):
   print("Saving CV to MongoDB:", structured_data)  # Print the structured data for debugging purposes
   cv_hash = hashlib.sha256(structured_data['cv_text'].encode()).hexdigest() # Generate a unique hash of the CV text using
SHA-256 to detect duplicates
   existing_cv = cv_collection.find_one({"cv_hash": cv_hash}) # Check if a CV with the same hash already exists in the
database
   # If a CV with the same hash exists, print a message and skip insertion
   if existing_cv:
      print("This CV already exists in the database.")
   else:
      structured_data['cv_hash'] = cv_hash # Add the hash to the structured data to store in the database
```

```
        structured_data['filename'] = filename # Store the filename in the structured data for reference
        cv_collection.insert_one(structured_data)# Insert the structured data into the MongoDB collection
        print("CV saved to the database.") # Print a confirmation message after successful insertion

# Function to extract text from PDFs using PyMuPDF
def extract_text_from_pdf(pdf_file):
    try:
        doc = fitz.open(stream=pdf_file.read(), filetype="pdf") # Open the PDF file from the binary stream (in-memory file object)
        return " ".join([page.get_text() for page in doc]) # Extract text from all pages and join them into a single string
    except Exception as e:
        print(f"Error extracting text from PDF: {str(e)}") # Catch any error during the PDF reading or text extraction process
        raise # Re-raise the exception so it can be handled by the caller

# Function to extract text from DOCX files
def extract_text_from_docx(docx_file):
    try:
        doc = docx.Document(docx_file) # Open the DOCX file using the docx.Document class
        return "\n".join([para.text for para in doc.paragraphs]) # Extract text from each paragraph and join them with newline
characters
    except Exception as e:
        print(f"Error extracting text from DOCX: {str(e)}") # Catch any error during DOCX reading or extraction
        raise # Re-raise the exception so it can be handled by the caller

# Function to extract contact information
def extract_contact_info(cv_text):
    email = re.findall(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0.-]+\.[A-Z|a-z]{2,}\b', cv_text) # Use a regex pattern to find email addresses
in the CV text
    phone = re.findall(r'\b\d{10,13}\b', cv_text)  # A basic pattern for phone numbers
    return {
        "email": email[0] if email else "Not Found", # Take the first matched email or "Not Found"
        "phone": phone[0] if phone else "Not Found"  # Take the first matched phone or "Not Found"
    }

# Function to extract education details simplified regex logic
def extract_education(cv_text):
    # List of keywords related to education that we want to search for in the CV text
    education_keywords = ["Bachelor", "Master", "PhD", "BSc", "MSc", "MBA", "BA", "BS", "MS", "University", "College", "Degree"]
    education_section = "" # Initialize an empty string to store extracted education details
    # Loop through each keyword in the list of education keywords
    for keyword in education_keywords:
        # Search for the keyword followed by any text until a newline character
        match = re.search(rf'{keyword}.*\n', cv_text, re.IGNORECASE)
        # If a match is found, add the matched text to the education_section
        if match:
            education_section += match.group() + '\n' # Append the matched result with a newline
    # Return the collected education details, removing any extra whitespace, otherwise return "Not Found"
    return education_section.strip() if education_section else "Not Found"

# Function to extract work experience
def extract_experience(cv_text):
    # List of keywords related to work experience that we want to search for in the CV text
    experience_keywords = ["Experience", "Work History", "Employment", "Projects", "Responsibilities", "Duties"]
    experience_section = "" # Initialize an empty string to store extracted experience details
    # Loop through each keyword in the list of experience keywords
    for keyword in experience_keywords:
        # Search for the keyword followed by multiple lines of text
        match = re.search(rf'{keyword}.*\n(.*\n)+', cv_text, re.IGNORECASE)
        # If a match is found, add the matched text to the experience_section
        if match:
            experience_section += match.group() + '\n' # Append the matched result with a newline
    # Return the collected experience details, removing any extra whitespace, otherwise return "Not Found"
    return experience_section.strip() if experience_section else "Not Found"

# Function to get skills from ESCO API
def get_skills_from_esco(job_title):
    esco_api_url = "https://ec.europa.eu/esco/api/search" # Define the base URL for the ESCO API search endpoint
    try:
        # Send a GET request to the ESCO API to search for the specified job title
        response = requests.get(
            esco_api_url,
            params={
                'text': job_title, # Specify the job title to search for
                'type': 'occupation', # Filter the search to look for occupations only
                'limit': 1  # Get the best match
            }
        )
        if response.status_code == 200: # Checks if the request was successful
            esco_data = response.json() # Converts the response to JSON format
            if esco_data and esco_data['results']: # Check if the response contains data and the 'results' field is populated
                occupation_uri = esco_data['results'][0]['uri'] # Extract the URI for the best-matching occupation

                # Fetch skills based on occupation URI
                skills_response = requests.get(f"https://ec.europa.eu/esco/api/resource/occupation/{occupation_uri}/skills")
                if skills_response.status_code == 200:
                    skills_data = skills_response.json() # Convert the skills response to JSON
                    return [skill['title'] for skill in skills_data] # Extract and return a list of skill titles from the JSON response
                else:
                    # Print an error message if the skills request fails
                    print(f"Error retrieving skills from ESCO: {skills_response.status_code}")
                    return [] # Returns empty list if skills retrieval fails
            else:
                print("No matching occupation found") # Print an error message if no matching occupation is found
                return [] # Return an empty list if no match is found
        else:
            # Print an error message if the occupation search request fails
            print(f"Error retrieving occupation from ESCO: {response.status_code}")
            return [] # Return an empty list if the request fails
    except Exception as e:
        print(f"Error calling ESCO API: {str(e)}") # Catch any exceptions that occur during the API calls
        return [] # Return an empty list if an exception occurs

def extract_skills(cv_text):
    global extracted_skills_ESCO # Access the global list of dynamically fetched skills
    skills = [] # Initialize an empty list to store matched skills
    # Predefined list of skills to search for in the CV text
    predefined_skills = [
# Programming Languages
    "python", "r", "sql", "java", "c++", "c#", "scala", "javascript", "typescript", "go",
    "ruby", "php", "kotlin", "swift", "rust", "bash", "powershell", "matlab", "objective-c",
    "perl", "dart", "haskell", "lua", "f#", "julia", "groovy", "shell scripting", "smalltalk",
# Machine Learning & AI
"tensorflow", "pytorch", "scikit-learn", "keras", "huggingface", "openai", "chatgpt",
"bert", "gpt-3", "llm", "transformers", "xgboost", "lightgbm", "catboost", "nltk",
"spacy", "deep learning", "cnn", "rnn", "gan", "vae", "autoencoders", "neural networks",
"natural language processing", "transfer learning", "fine-tuning", "reinforcement learning",
"graph neural networks", "contrastive learning", "clustering", "classification",
"anomaly detection", "generative ai", "few-shot learning", "self-supervised learning",
"semi-supervised learning", "zero-shot learning", "bayesian networks","data analysis",
"data preprocessing",

# Data Science & Analytics
"pandas", "numpy", "scipy", "matplotlib", "seaborn", "plotly", "dash", "statsmodels",
"shap", "lime", "mlflow", "kubeflow", "feature engineering", "data cleaning",
"data wrangling", "time series analysis", "statistical modeling", "hypothesis testing",
"ab testing", "anova", "decision trees", "random forests", "support vector machines",
"gradient boosting", "dimensionality reduction", "feature extraction", "data pipelines",

# Cloud Platforms & Services
"aws", "azure", "gcp", "ibm cloud", "oracle cloud", "aws lambda", "aws ec2", "aws s3",
"bigquery", "cloud run", "cloud functions", "vertex ai", "aws glue", "redshift",
"cloudwatch", "azure synapse", "azure databricks", "blob storage", "aws sagemaker",
"aws cloudformation", "terraform", "kubernetes", "eks", "aks", "gke", "helm",
"elastic beanstalk", "azure app service", "firebase", "cloud storage", "cloud security",
"cloud migration", "hybrid cloud", "multi-cloud",

# Web Development
"html5", "css3", "sass", "less", "bootstrap", "tailwind", "javascript", "jquery",
"react", "angular", "vue.js", "next.js", "nuxt.js", "svelte", "astro", "django",
"flask", "fastapi", "express", "nestjs", "hapi.js", "graphql", "apollo",
"socket.io", "rest api", "soap", "grpc",

# Backend Development
"node.js", "spring boot", "express.js", "django", "flask", "fastapi", "rust actix",
"go fiber", "java servlet", "php laravel", "ruby on rails", "gin", "asp.net",

# Frontend Frameworks
"react", "angular", "vue", "svelte", "solidjs", "preact", "next.js", "nuxt.js",
"tailwind", "bootstrap", "material ui", "chakra ui",

# Mobile Development
"react native", "flutter", "kotlin", "swift", "xamarin", "phonegap", "capacitor",
"android studio", "xcode",

# DevOps & Automation
"docker", "kubernetes", "terraform", "ansible", "puppet", "chef", "jenkins", "circleci",
"travis ci", "gitlab ci/cd", "github actions", "argo cd", "vault", "helm", "docker compose",
"spinnaker", "prometheus", "grafana", "splunk", "datadog",

# Networking & Security
"tcp/ip", "udp", "dns", "vpn", "firewall", "load balancing", "ssl/tls", "https",
"http/2", "http/3", "ipv4", "ipv6", "proxy", "nginx", "apache", "cisco ios",
"bgp", "ospf", "eigrp", "wireshark", "pfSense", "ipsec", "openvpn", "cloudflare",

# Cybersecurity
```

```
    "penetration testing", "ethical hacking", "metasploit", "nessus", "burp suite",
    "red team", "blue team", "vulnerability scanning", "threat hunting", "malware analysis",
    "siem", "soc", "snort", "cylance", "crowdstrike", "carbon black", "darktrace",
    "ssl inspection", "sandboxing", "network segmentation", "threat intelligence",
    "security orchestration", "ai-driven threat detection", "incident response",
    "encryption protocols", "firewall configuration", "ids/ips", "risk assessment",
    "patch management", "endpoint protection", "splunk", "qualys", "wireshark", "cisco",
    "security auditing", "active directory", "group policy",

    # Database Management
    "mysql", "postgresql", "mongodb", "redis", "sqlite", "oracle db", "ms sql server",
    "couchbase", "neo4j", "elasticsearch", "dynamodb", "amazon aurora", "firestore",
    "influxdb", "timescaledb",

    # Testing & QA
    "selenium", "cypress", "playwright", "junit", "pytest", "testng", "mocha",
    "jasmine", "karma", "appium", "loadrunner", "postman", "rest assured",

    # Operating Systems
    "linux", "ubuntu", "debian", "red hat", "centos", "kali linux", "windows server",
    "macos", "unix", "freebsd", "openbsd",

    # Blockchain & Cryptography
    "ethereum", "solidity", "bitcoin", "hyperledger", "cardano", "defi", "zero knowledge proofs",
    "cryptography", "openssl", "jwt", "sha256", "elliptic curve",

    # Data Formats & Parsing
    "json", "xml", "csv", "parquet", "orc", "yaml", "protobuf", "avro",

    # Search & Indexing
    "elasticsearch", "solr", "opensearch", "algolia", "redisearch", "vector search",
    "k-nearest neighbors",

    # Visualization & BI
    "tableau", "power bi", "looker", "metabase", "superset", "google data studio",
    "dash", "plotly", "seaborn", "bokeh",

    # Game Development
    "unity", "unreal engine", "godot", "game physics", "rendering", "shader programming",

    # Mathematical & Statistical Skills
    "linear algebra", "probability", "statistics", "calculus", "bayesian inference",
    "game theory", "graph theory",

    # Additional Skills
    "pytorch lightning", "scalability", "high availability", "distributed systems",
    "message queues", "event-driven architecture", "api gateway", "microservices",
    "load balancing", "caching", "rate limiting", "feature flags"
]

    all_skills = list(set(predefined_skills + extracted_skills_ESCO))  # Remove duplicates
    cv_text = re.sub(r'\s+', ' ', cv_text).strip().lower() # Clean up text by removing extra spaces and normalizing cases
    # Clean up each skill in the list of all skills
    for i in range(len(all_skills)):
        # Remove extra spaces and normalize case for consistent matching
        all_skills[i] = re.sub(r'\s+', ' ', all_skills[i]).strip().lower()

    # Use fuzzy matching to extract the best possible matches
    # `process.extract()` returns the top matches based on similarity score
    # `fuzz.token_set_ratio` allows matching even if the order of words is different
    potential_matches = process.extract(cv_text, all_skills, scorer=fuzz.token_set_ratio, limit=20)

    for match, score in potential_matches:
        # Lower threshold for multi-word terms to allow flexible matching
        if ' ' in match:
            threshold = 65  # Lower threshold for multi-word terms to account for small variations
        else:
            threshold = 70 # Higher threshold for single-word terms for stricter matching
        # Only consider matches that exceed the similarity threshold
        if score > threshold:
            # Length penalty – avoid partial or unrelated matches by limiting length difference
            # Allowing a buffer of 10 characters to account for reasonable length differences
            if len(match) <= len(cv_text) + 10:
                # First context check:
                # Direct word boundary check to avoid false positives from partial words
                # Partial ratio allows for substring-based similarity checks
                if f" {match} " in f" {cv_text} " or fuzz.partial_ratio(match, cv_text) > 80:
                    skills.append(match)

                # Second context check:
                # Split the multi-word terms and try to match individual words using partial_ratio
                # Allows catching skills embedded in longer or complex sentences
                elif any(fuzz.partial_ratio(word.lower(), cv_text.lower()) > 80 for word in match.split()):
                    skills.append(match)

    return skills if skills else "Not Found" # Return matched skills if found; otherwise, return "Not Found"

# Function to organize CV data
def organize_cv_data(cv_text):
    return {
        "contact_info": extract_contact_info(cv_text), # Extract contact information (email and phone) from the CV text
        "education": extract_education(cv_text), # Extract educational background from the CV text
        "experience": extract_experience(cv_text), # Extract work experience details from the CV text
        "skills": extract_skills(cv_text), # Extract skills listed in the CV text
        "cv_text": cv_text  # Full CV text is also stored in case something is not picked up
    }

# Function to calculate weighted skill score
def calculate_skill_score(user_skills, game_skills):
    total_score = 0 # Initialize the total score to zero
    # Loop through each skill and its weight in the game_skills dictionary
    for skill, weight in game_skills.items():
        # Check if the current skill is present in the user's skill set
        if skill in user_skills:
            # If the skill is matched, add the weighted score (scaled to percentage) to the total score
            total_score += weight * 100  # Scale up the weights to percentages
    return total_score # Return the final calculated score after the loop finishes

# Home route
@app.route('/')
```

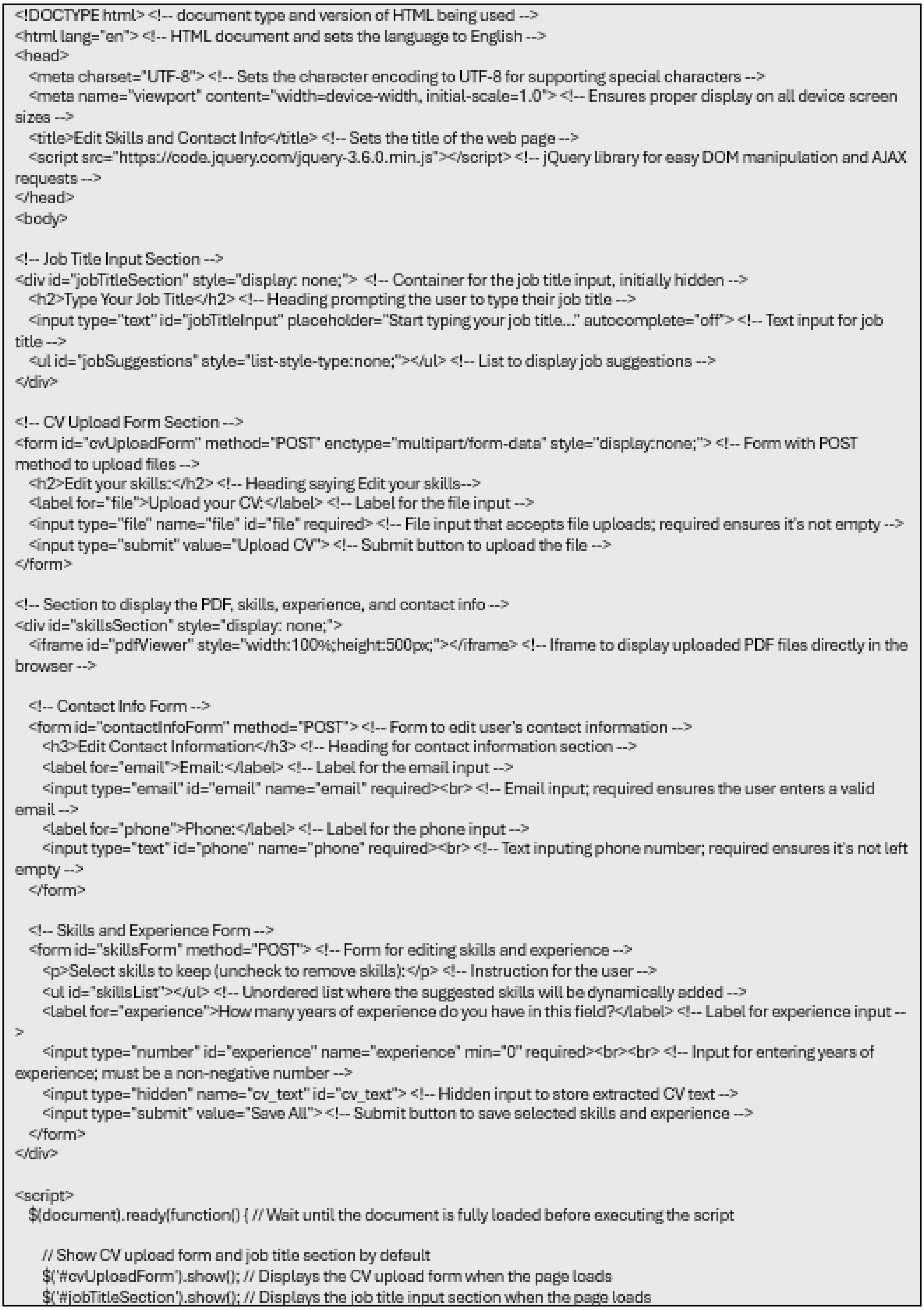
```
# home function that renders the main page
def home():
   return render_template('LSBUPhase1.html') #LSBUPhase1.html

#--------------------------------------------------------------------------------orginal Upload CV method
# Route to upload CV
@app.route('/upload_cv', methods=['POST']) # Defines a route '/upload_cv' that listens for POST requests
def upload_cv():
   uploaded_file = request.files['file'] # Retrieves the uploaded file from the form data under the key 'file'

   if uploaded_file:  # Checks if a file was uploaded
      try:
         # Save the file temporarily
         file_path = os.path.join(UPLOAD_FOLDER, uploaded_file.filename) # Constructs the full file path using the upload folder
and filename
         uploaded_file.save(file_path) # Saves the uploaded file to the constructed path
         file_extension = uploaded_file.filename.split('.')[-1].lower() # Extracts the file extension (e.g., pdf, docx) and converts to
lowercase
         print(f"Received file: {uploaded_file.filename}")  # Debugging print
         if file_extension == 'pdf': # checks if the file is a PDF
            with open(file_path, 'rb') as pdf_file: # Open the PDF file in binary read mode
               cv_text = extract_text_from_pdf(pdf_file) # Extract text from the PDF file using the extract_text_from_pdf() function
            print("PDF processed successfully") # Debugging print to confirm successful PDF processing
         elif file_extension == 'docx': # checks if the file is a DOCX
            with open(file_path, 'rb') as docx_file: # Open the DOCX file in binary read mode
               cv_text = extract_text_from_docx(docx_file) # Extract text from the DOCX file using the extract_text_from_docx()
function
            print("DOCX processed successfully") # Debugging print to confirm successful DOCX processing
         else:
            print("Unsupported file format") # Debugging print to indicate unsupported format
            return jsonify({'error': 'Unsupported file format'}), 400 # Return an error message with the status code 400

         processed_cv_text = preprocess_text(cv_text) # Clean and preprocess the extracted CV text
         structured_data = organize_cv_data(processed_cv_text) # Organize the extracted data such as skills, contact info, and
more

         print("Processed CV data:", structured_data)  # Debugging print

         pdf_url = url_for('view_pdf', filename=uploaded_file.filename) # Generate a URL to view the uploaded file
         print(f"PDF URL: {pdf_url}")  # Debugging print, shows the pdf URL

         return jsonify({ # Return the extracted data as a JSON response
            'skills': structured_data['skills'], # Include extracted skills
            'cv_text': cv_text, # Include unmodified CV text
            'pdf_url': pdf_url,  # Return the PDF URL
            'contact_info': structured_data['contact_info']  # Include contact information
         })
      except Exception as e: # Handle any errors that occur during processing
         print(f"Error processing the file: {str(e)}")  # Debugging print to display the error message
         return jsonify({'error': f'Error processing the file: {str(e)}'}), 500 # Return error message with the status code 500
   else:
      print("No file uploaded")  # Debugging print, to help indicate file is not uploaded
      return jsonify({'error': 'No file uploaded'}), 400 # Return an error message with the status code 400

@app.route('/search_jobs', methods=['GET']) # Defines a Flask route at the endpoint '/search_jobs' to handle GET requests
```

```
def search_jobs():
    job_title = request.args.get('job_title') # Retrieves the 'job_title' parameter from the request URL
    esco_api_url = "https://ec.europa.eu/esco/api/search" # URL for the ESCO API endpoint

    try:
        # Call ESCO API to search for job titles
        response = requests.get( # Sends a GET request to the ESCO API
            esco_api_url,
            params={
                'text': job_title, # Passes the job title as a query parameter to the API
                'type': 'occupation', # Specifies that the API should search for occupations
                'limit': 5  # Return top 5 matches
            }
        )

        # Debugging: Print the API response or status code
        if response.status_code == 200: # Checks if the API response is successful
            esco_data = response.json() # Parses the JSON response into a Python dictionary
            print("ESCO Job search response:", esco_data)  # Debugging print

            # Ensure we access the correct path to 'results'
            if '_embedded' in esco_data and 'results' in esco_data['_embedded']: # Checks if the 'results' key exists in the response
                jobs = [result['title'] for result in esco_data['_embedded']['results']] # Extracts job titles from the API response
                return jsonify({'jobs': jobs}) # Returns the list of job titles as a JSON response
            else:
                return jsonify({'error': 'No results found in ESCO API'}), 500  # Returns an error if no results are found
        else:
            print(f"Error in job search, status code: {response.status_code}")  # Debugging print shows the error status code
            return jsonify({'error': 'Error retrieving jobs from ESCO API'}), 500 # Returns an error message with status code 500
    except Exception as e:
        print(f"Error calling ESCO API: {str(e)}")  # Debugging print to display the exception details
        return jsonify({'error': str(e)}), 500 # Returns an error message with status code 500 if an exception occurs

@app.route('/get_job_skills', methods=['GET'])  # Defines a Flask route at the endpoint '/get_job_skills' to handle GET requests
def get_job_skills():
    global extracted_skills_ESCO  # Declare that we are using the global variable
    job_title = request.args.get('job_title') # Get the 'job_title' parameter from the request URL

    # Step 1: Search for the job/occupation using searchGet API
    search_url = f"https://ec.europa.eu/esco/api/search?text={job_title}&type=occupation&limit=5" # Format the search URL
with the job title and set the limit to 5 results
    search_response = requests.get(search_url) # Send a GET request to the ESCO API search endpoint

    if search_response.status_code == 200: # Check if the search request was successful
        search_results = search_response.json() # Convert the JSON response to a Python dictionary
        if '_embedded' in search_results and 'results' in search_results['_embedded']: # Check if 'results' exists in the response
            occupation = search_results['_embedded']['results'][0] # Take the first occupation from the search results
            occupation_uri = occupation['uri'] # Extract the URI of the selected occupation
            print(f"Occupation URI: {occupation_uri}") # Output the occupation URI to the console
            skill_api_url = "https://ec.europa.eu/esco/api/resource/skill" # Define the ESCO skill API endpoint
            skill_response = requests.get(skill_api_url, params={'uri': occupation_uri}) # Send a GET request to the skill API using the
occupation URI

            # Debugging print if needed
            print(f"Fetching skills from: {skill_api_url}?uri={occupation_uri}") # Output the full skill request URL
        print(f"Skills Response Status Code: {skill_response.status_code}") # Output the status code of the skill request

        if skill_response.status_code == 200: # Check if the skill request was successful
          # Extract and deduplicate titles
          titles = re.findall(r'"title":\s*"([^"]+)"', skill_response.content.decode('utf-8')) # Find all skill titles using regex
          extracted_skills_ESCO = list(set(titles))  # Remove duplicates and store as a list

          print("\nExtracted Titles:", extracted_skills_ESCO)  # Output deduplicated titles to the terminal

          skills_data = skill_response.json() # Convert the skill response to JSON
          return jsonify({
            'skills_data': skills_data,  # Include the raw JSON if necessary
            'extracted_titles': extracted_skills_ESCO   # Titles extracted for skill comparison
          })
        else:
          print(f"Error fetching skills: {skill_response.status_code}") # Output an error message if skill fetching fails
          return jsonify({'error': 'Error fetching job skills'}), 500 # Return an error response with status code 500
      else:
        return jsonify({'error': 'No occupations found'}), 404 # Return a 404 error if no occupations are found in the search results
    else:
      return jsonify({'error': 'Error fetching job suggestions'}), 500 # Return a 500 error if the search request fails

# Route to view PDF
@app.route('/view_pdf/<filename>') # Defines a Flask route at the endpoint '/view_pdf/<filename>' to serve PDF files
def view_pdf(filename):
  file_path = os.path.join(UPLOAD_FOLDER, filename) # file path using the upload folder and the provided filename
  if os.path.exists(file_path): # Checks if the file exists at the specified path
    try:
      return send_file(file_path) # If the file exists, serve it to the client using Flask's send_file() function
    except Exception as e: # Catch any exception that occurs while serving the file
      print(f"Error serving PDF file: {str(e)}")  # Debugging print to display the exception details
      return jsonify({'error': str(e)}) # Return a JSON response with the error message
  else:
    print(f"File not found: {filename}")  # Debugging print to indicate that the file was not found
    return jsonify({'error': f'File {filename} not found'}), 404 # Return a 404 error if the file doesnt  exist

# Route to test the MongoDB connection
@app.route('/test_mongodb')  # Defines a Flask route at the endpoint '/test_mongodb'
def test_mongodb(): # Function to handle the request to the '/test_mongodb' route
  try:
    cv_count = cv_collection.count_documents({}) # Count the number of documents in the 'cv_collection' collection
    return f'MongoDB connected. {cv_count} CVs found in the database.' # Return a success message with the number of CVs
found
  except Exception as e:
    return f'Error connecting to MongoDB: {str(e)}' # If an error occurs, return an error message with the exception details

# Route to process skills and save them
@app.route('/process_skills', methods=['POST']) # Defines a Flask route at the endpoint '/process_skills' to handle POST
requests
def process_skills():
  skills = request.form.getlist('skills') # Extracts the list of skills from the form data
  cv_text = request.form['cv_text'] # Extracts the raw CV text from the form data
  years_of_experience = request.form['experience'] # Extracts the years of experience from the form data

  # Extracts contact information (email and phone) from the form data and stores it in a dictionary
  contact_info = {
    'email': request.form['email'], # Extracts the email address
    'phone': request.form['phone'] # Extracts the phone number
  }
  structured_data = organize_cv_data(cv_text) # Organizes the extracted data into a structured format using the
organize_cv_data() function
  structured_data['skills'] = skills # Adds the extracted skills to the structured data
  structured_data['years_of_experience'] = years_of_experience # Adds the years of experience to the structured data
  structured_data['contact_info'] = contact_info # Adds the contact info email and phone to the structured data

  # Saves the structured data to MongoDB using the save_cv_to_mongodb() function
  save_cv_to_mongodb("user_uploaded_cv", structured_data)

  # Returns a JSON response confirming that the data was saved successfully
  return jsonify({'message': 'Skills, experience, and contact information saved successfully'})

# Route to display games and calculate skill match
@app.route('/games')
def games():
  games_data = [ # Define the list of available games with their required skills, image, and required score to unlock
    {
      "name": "Website Quiz", # title of the game/quiz/survey
      "skills_required": {"C++": 0.5, "Java": 0.5}, # Skills required and their weights for the game
      "image": "https://www.stx.ox.ac.uk/sites/default/files/stx/images/article/depositphotos_41197145-stock-photo-
quiz.jpg", # URL for the game's image
      "required_score": 75 # Minimum score required to unlock the game
    },
    {
      "name": "Ai Survey", # title of the game/quiz/survey
      "skills_required": {"machine learning": 0.25, "python": 0.50, "tensorflow": 0.25}, # Skills required and their weights for the
game
      "image": "https://www.questionpro.com/blog/wp-content/uploads/2024/02/AI-Survey.jpg",# URL for the game's image
      "required_score": 75 # Minimum score required to unlock the game
    },
    {
      "name": "Survey", # title of the game/quiz/survey
      "skills_required": {"React": 0.5, "JavaScript": 0.5}, # Skills required and their weights for the game
      "image": "https://img.freepik.com/free-vector/online-survey-tablet_3446-296.jpg", # URL for the game's image
      "required_score": 75 # Minimum score required to unlock the game
    },
    {
      "name": "Research Survey", # title of the game/quiz/survey
      "skills_required": {"docker": 0.5, "aws": 0.5}, # Skills required and their weights for the game
      "image": "https://www.aimtechnologies.co/wp-content/uploads/2024/02/Types-of-Survey-Research.jpg", # URL for the
game's image
      "required_score": 75 # Minimum score required to unlock the game
    }
  ]
  user_cv = cv_collection.find_one(sort=[("_id", -1)])  # Get the latest uploaded CV
  # Extract the list of user skills from the CV and convert them to lowercase for comparison
  user_skills = [skill.lower() for skill in user_cv.get("skills", [])] if user_cv else []
  response_data = [] # Initialize an empty list to store game results
  # Loop through each game and calculate the user's eligibility based on skills
  for game in games_data:
    game_skills = list(game['skills_required'].keys()) # Get the list of required skills for the game
```

```
        # Calculate skills the user has and is missing
        matched_skills = [skill for skill in game_skills if skill in user_skills] # Identify matched skills
        missing_skills = [skill for skill in game_skills if skill not in user_skills] # Identify missing skills

        # Calculate user's score based on the matched skills
        user_score = calculate_skill_score(matched_skills, game['skills_required'])
        can_play = user_score >= game['required_score'] # Determine if the user can unlock the game based on their score
        # Set the game status based on the user's score
        game_status = "You can Take this Survey/Quiz!" if can_play else f"You need {game['required_score'] - user_score}% more
to unlock this Survey/Quiz."

        # Append the calculated game info to the response list
        response_data.append({
            "name": game['name'], # Name of the game
            "image": game['image'], # URL for the game's image
            "status": game_status, # Display message based on the user's score
            "user_score": user_score, # User's calculated score
            "required_score": game['required_score'], # Score needed to unlock the game
            "matched_skills": matched_skills, # List of matched skills
            "missing_skills": missing_skills # List of missing skills
        })
    return jsonify({"games": response_data, "user_skills": user_skills}) # Return the game data and user skills as a JSON
response

# Route to get game details
@app.route('/game/<game_name>')
def game_detail(game_name):
    game_details = { # Define the detailed information for each game
        "Website Quiz": {
            "skills_required": ["C++", "Java"], # Skills required for this game
            "description": "A web development game that involves coding in C++ and Java.", # Description of the game
            "image": "https://www.stx.ox.ac.uk/sites/default/files/stx/images/article/depositphotos_41197145-stock-photo-
quiz.jpg" # URL for the game's image
        },
        "Ai Survey": {
            "skills_required": ["machine learning", "python", "tensorflow"], # Skills required for this game
            "description": "An AI-based survey tool requiring Python, TensorFlow, and machine learning expertise.", # Description of
the game
            "image": "https://devskrol.com/wp-content/uploads/2021/10/PythonQ1S2-1.jpg" # URL for the game's image
        },
        "Survey": {
            "skills_required": ["React", "JavaScript"], # Skills required for this game
            "description": "A card-based game where you need React and JavaScript skills to develop the front end.", # Description of
the game
            "image": "https://img.freepik.com/free-vector/online-survey-tablet_3446-296.jpg" # URL for the game's image
        },
        "Research Survey": {
            "skills_required": ["docker", "aws"], # Skills required for this game
            "description": "A space simulation game where Docker and AWS are required to handle cloud operations.", # Description
of the game
            "image": "https://www.aimtechnologies.co/wp-content/uploads/2024/02/Types-of-Survey-Research.jpg" # URL for the
game's image
        }
    }
    # Check if the requested game exists in the dictionary
    if game_name in game_details:
        game_info = game_details[game_name] # Fetch the game details
        game_info["matched_skills"] = []  # Add matched skills here if available
        game_info["missing_skills"] = []  # Add missing skills here if available
        return jsonify(game_info) # Return the game details as a JSON response
    else:
        return jsonify({"error": "Game not found"}), 404 # Return an error message if the game is not found

# Email-related functions for reminders
def get_user_emails():
    user_emails = [] # Initialize an empty list to store user emails
    for cv in cv_collection.find({}, {'contact_info.email': 1}): # Query MongoDB to get the 'email' field from the 'contact_info'
document
        email = cv.get('contact_info', {}).get('email', None) # Get the email field; return None if not found
        if email:
            user_emails.append(email) # If email exists, add it to the list
    return user_emails # Return the list of emails

def send_email(recipient_email, subject, message):
    smtp_server = "smtp.gmail.com" # Define the SMTP server for Gmail
    smtp_port = 587 # Define the SMTP port for Gmail
    smtp_user = "email@gmail.com"  # Sender's email address
    smtp_password = "Password"  # Sender's email password App Password

    msg = MIMEMultipart()
    msg['From'] = smtp_user # Set the sender's email address
    msg['To'] = recipient_email # Set the recipient's email address
    msg['Subject'] = subject # Set the subject of the email
    msg.attach(MIMEText(message, 'plain')) # Attach the message body as plain text

    try:
        server = smtplib.SMTP(smtp_server, smtp_port) # Connect to the SMTP server
        server.starttls() # Start TLS encryption for a secure connection
        server.login(smtp_user, smtp_password) # Log into the SMTP server using credentials
        server.send_message(msg) # Send the email message
        server.quit() # Close the connection to the server
        print(f"Email sent to {recipient_email}") # Print a success message
    except Exception as e:
        print(f"Failed to send email to {recipient_email}: {e}") # Print an error message if sending fails

# Function to create the reminder message
def create_email_message():
    link_to_games = "http://127.0.0.1:5001/games" # URL link to the games page
    message = f"""
    Hello,

    This is a reminder that new quizs/surveys and more are out click on the link below to view them:

    {link_to_games}

    Best regards,
    Your Game Platform Team
    """
    return message # Return the formatted message
```

```
# Function to send reminders to all users
def send_reminders():
    user_emails = get_user_emails() # Get the list of user emails from MongoDB
    message = create_email_message() # Create the email message
    subject = "New Games Unlocked - Check them out!"  # Define the email subject

    for email in user_emails: # Loop through all user emails
        send_email(email, subject, message) # Send the reminder email to each user

# Schedule email reminders every Wednesday 3:00pm
schedule.every().wednesday.at("15:00").do(send_reminders) # Use the schedule library to set up automatic reminders

print("Scheduler is running...") # for debugging

# Run the scheduler
def run_scheduler():
    while True:
        schedule.run_pending() # Check if any scheduled tasks are ready to run
        time.sleep(60) # Wait for 60 seconds before checking again

# Start the scheduler in a separate thread if needed, or directly in __main__
if __name__ == "__main__":
    import threading # Import threading to run scheduler in parallel
    scheduler_thread = threading.Thread(target=run_scheduler) # Create a thread for the scheduler
    scheduler_thread.start() # Start the scheduler thread
    app.run(debug=True, port=5001) # Start the Flask app on port 5001 in debug mode
```

Appendix A.2 – Phase 1: Skill Extraction Logic Frontend



```
<!DOCTYPE html> <!-- document type and version of HTML being used -->
<html lang="en"> <!-- HTML document and sets the language to English -->
<head>
   <meta charset="UTF-8"> <!-- Sets the character encoding to UTF-8 for supporting special characters -->
   <meta name="viewport" content="width=device-width, initial-scale=1.0"> <!-- Ensures proper display on all device screen
sizes -->
   <title>Edit Skills and Contact Info</title> <!-- Sets the title of the web page -->
   <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script> <!-- jQuery library for easy DOM manipulation and AJAX
requests -->
</head>
<body>

<!-- Job Title Input Section -->
<div id="jobTitleSection" style="display: none;">  <!-- Container for the job title input, initially hidden -->
   <h2>Type Your Job Title</h2> <!-- Heading prompting the user to type their job title -->
   <input type="text" id="jobTitleInput" placeholder="Start typing your job title..." autocomplete="off"> <!-- Text input for job
title -->
   <ul id="jobSuggestions" style="list-style-type:none;"></ul> <!-- List to display job suggestions -->
</div>

<!-- CV Upload Form Section -->
<form id="cvUploadForm" method="POST" enctype="multipart/form-data" style="display:none;"> <!-- Form with POST
method to upload files -->
   <h2>Edit your skills:</h2> <!-- Heading saying Edit your skills-->
   <label for="file">Upload your CV:</label> <!-- Label for the file input -->
   <input type="file" name="file" id="file" required> <!-- File input that accepts file uploads; required ensures it's not empty -->
   <input type="submit" value="Upload CV"> <!-- Submit button to upload the file -->
</form>

<!-- Section to display the PDF, skills, experience, and contact info -->
<div id="skillsSection" style="display: none;">
   <iframe id="pdfViewer" style="width:100%;height:500px;"></iframe> <!-- Iframe to display uploaded PDF files directly in the
browser -->

   <!-- Contact Info Form -->
   <form id="contactInfoForm" method="POST"> <!-- Form to edit user's contact information -->
      <h3>Edit Contact Information</h3> <!-- Heading for contact information section -->
      <label for="email">Email:</label> <!-- Label for the email input -->
      <input type="email" id="email" name="email" required><br> <!-- Email input; required ensures the user enters a valid
email -->
      <label for="phone">Phone:</label> <!-- Label for the phone input -->
      <input type="text" id="phone" name="phone" required><br> <!-- Text inputing phone number; required ensures it's not left
empty -->
   </form>

   <!-- Skills and Experience Form -->
   <form id="skillsForm" method="POST"> <!-- Form for editing skills and experience -->
      <p>Select skills to keep (uncheck to remove skills):</p> <!-- Instruction for the user -->
      <ul id="skillsList"></ul> <!-- Unordered list where the suggested skills will be dynamically added -->
      <label for="experience">How many years of experience do you have in this field?</label> <!-- Label for experience input --
>
      <input type="number" id="experience" name="experience" min="0" required><br><br> <!-- Input for entering years of
experience; must be a non-negative number -->
      <input type="hidden" name="cv_text" id="cv_text"> <!-- Hidden input to store extracted CV text -->
      <input type="submit" value="Save All"> <!-- Submit button to save selected skills and experience -->
   </form>
</div>

<script>
   $(document).ready(function() { // Wait until the document is fully loaded before executing the script

      // Show CV upload form and job title section by default
      $('#cvUploadForm').show(); // Displays the CV upload form when the page loads
      $('#jobTitleSection').show(); // Displays the job title input section when the page loads
```

```
// Hide the purpose section (if it's still in the code or temporarily commented out)
$('#purposeSection').hide(); // Hides the purpose section if it exists

// Handle input for job title
$('#jobTitleInput').on('input', function() { // Add an event listener for the job title input field
  var jobTitle = $(this).val(); // Get the value entered by the user in the job title field
  if (jobTitle.length > 2) {  // Start fetching suggestions after 3 characters
    $.ajax({
      url: '/search_jobs', // Endpoint to search for job titles
      type: 'GET', // Send a GET request to the server
      data: { job_title: jobTitle }, // Pass the user-entered job title as a parameter
      success: function(response) { // Function to execute if the request is successful
        var jobSuggestions = $('#jobSuggestions'); // Get the list where suggestions will be shown
        jobSuggestions.empty(); // Clear previous suggestions
        if (response.jobs && response.jobs.length > 0) { // Check if jobs are returned in the response
          response.jobs.forEach(function(job) { // Loop through the jobs array in the response
            jobSuggestions.append('<li>' + job + '</li>'); // Append each job suggestion as a list item
          });
        } else {
          jobSuggestions.append('<li>No jobs found</li>'); // Show message if no jobs are found
        }
      },
      error: function() {
        alert('Error fetching job suggestions. Please try again.'); // Show an error alert if request fails
      }
    });
  }
});

// Handle job selection and fetch skills
$(document).on('click', '#jobSuggestions li', function() { // Add a click event listener to the job suggestions list items
  var selectedJob = $(this).text(); // Get the text content of the clicked suggestion
  $('#jobTitleInput').val(selectedJob); // Set the input value to the selected job title
  $('#jobSuggestions').empty(); // Clear the suggestions list after selecting a job

  $.ajax({
    url: '/get_job_skills', // Endpoint to retrieve skills for the selected job
    type: 'GET', // Send a GET request to the server
    data: { job_title: selectedJob }, // Pass the selected job title as a parameter
    success: function(response) { // Function to execute if the request is successful
      var skillsList = $('#skillsList'); // Get the list where skills will be shown
      skillsList.empty(); // Clear previous skills
      if (response.skills && response.skills.length > 0) { // Check if skills are returned in the response
        response.skills.forEach(function(skill) { // Loop through the skills array in the response
          skillsList.append('<li><input type="checkbox" name="skills" value="' + skill + '" checked> ' + skill + '</li>');
        });
      } else {
        skillsList.append('<li>No skills found for this job</li>'); // Show message if no skills are found
      }
    },
    error: function() { // Function to execute if the request fails
      alert('Error fetching job skills.'); // Show an error alert if request fails
    }
  });
});

// Handle CV upload form submission
$('#cvUploadForm').on('submit', function(event) { // Attaches a 'submit' event handler to the form with id 'cvUploadForm'
  event.preventDefault(); // Prevents the default form submission behavior basically it prevents page reload
  var formData = new FormData(this); // Creates a FormData object from the form data which is used for file uploads

  $.ajax({
    url: '/upload_cv', // Specifies the endpoint where the form data will be sent
```

```
          type: 'POST', // Sends the data using the HTTP POST method
          data: formData, // Passes the FormData object as the data to be sent
          contentType: false, // Prevents jQuery from setting a content-type header (important for file uploads)
          processData: false, // Prevents jQuery from processing the data into a query string
          success: function(response) { // Callback function that executes if the request succeeds
            if (response.error) { // Checks if the response contains an error message
              alert(response.error); // Displays the error message in an alert box
              return; // Stops further execution if an error is present
            }
            $('#email').val(response.contact_info.email); // Sets the value of the #email input to the extracted email from the
response
            $('#phone').val(response.contact_info.phone); // Sets the value of the #phone input to the extracted phone from the
response
            $('#skillsList').empty(); // Clears any existing list items from the #skillsList element
            response.skills.forEach(function(skill) { // Loops through each skill in the response.skills array
              $('#skillsList').append('<li><input type="checkbox" name="skills" value="' + skill + '" checked> ' + skill + '</li>');
              // Creates a list item with a checked checkbox for each skill and appends it to #skillsList
            });
            $('#cv_text').val(response.cv_text); // Sets the value of the #cv_text input to the extracted CV text from the response
            $('#pdfViewer').attr('src', response.pdf_url); // Sets the source URL of the #pdfViewer iframe to the extracted PDF
URL from the response
            $('#skillsSection').show(); // Displays the section with id #skillsSection which was previously hidden
            $('#cvUploadForm').hide(); // Hides the form with id #cvUploadForm after successful submission
          },
          error: function() { // Callback function that executes if the request fails
            alert('Error uploading CV. Please try again.'); // Displays an error message in a alert box
          }
        });
      });
    });

  </script>

<!-- New Section to act as the empty page (hidden initially) -->
<div id="emptyPageSection" style="display: none;">
  <h1>Relevant Surveys/Quizzes to your CV</h1> <!-- Main heading to inform the user about the purpose of the section -->
  <p>Below are all the Surveys/Quizzes that you can take. The locked icons are Surveys/Quizzes you do not qualify to take
yet.</p>
  <!-- Description explaining how the surveys/quizzes work and the concept of locked content -->
  <div id="gamesContainer" style="display: flex; flex-wrap: wrap;">
    <!-- Games will be dynamically inserted here -->
  </div>
</div>

<!-- Game Details Section (Hidden initially) -->
<div id="gameDetailSection" style="display:none;">
  <h2 id="gameName"></h2> <!-- Placeholder for the game's name, which will be dynamically inserted -->
  <img id="gameImage" src="" alt="Game Image" style="width:200px;"><br> <!-- Placeholder for the game image -->
  <p id="gameDescription"></p> <!-- Placeholder for the game's description, which will be dynamically set -->
  <p><strong>Required Skills:</strong> <span id="gameSkills"></span></p>
  <button id="backToGames">Back to Games</button> <!-- Button that allows the user to return to the list of available games
-->
</div>

<script>

  // Show CV upload form and job title section by default
  $('#cvUploadForm').show();
  $('#jobTitleSection').show();

  // Hide the purpose section (if it's still in the code or temporarily commented out)
  $('#purposeSection').hide();

  $('#jobTitleInput').on('input', function() { // Attaches an event handler to the 'input' event on the #jobTitleInput element
    var jobTitle = $(this).val(); // Retrieves the current value of the input field and stores it in the jobTitle variable
    if (jobTitle.length > 2) {  // Start fetching suggestions after 3 characters
        $.ajax({
            url: '/search_jobs', // Specifies the endpoint to send the request to
            type: 'GET', // Sends the request using the HTTP GET method
            data: {job_title: jobTitle}, // Sends the input value as a query parameter named 'job_title'
            success: function(response) { // Callback function that executes if the request succeeds
                var jobSuggestions = $('#jobSuggestions'); // Selects the #jobSuggestions element and stores it in the jobSuggestions
variable
                jobSuggestions.empty(); // Clears any existing suggestions in the #jobSuggestions element
                if (response.jobs && response.jobs.length > 0) { // Checks if the response contains jobs and if the jobs array is not
empty
                    response.jobs.forEach(function(job) { // Loops through each job in the jobs array
                        jobSuggestions.append('<li>' + job + '</li>'); // Appends each job as a new <li> element to the #jobSuggestions list
                    });
                } else { // Executes if the jobs array is empty or not present
                    jobSuggestions.append('<li>No jobs found</li>');  // Handle case where no jobs are found
                }
                console.log("Job suggestions:", response);  // Debugging print
            },
            error: function() {
                alert('Error fetching job suggestions. Please try again.'); // Displays an alert message if an error occurs
                console.error("Error fetching job suggestions");  // Debugging print
            }
        });
    }
});

// Handle job selection and fetch skills
$(document).on('click', '#jobSuggestions li', function() {
    var selectedJob = $(this).text(); // Get the text value of the selected list item
    $('#jobTitleInput').val(selectedJob); // Set the job title input field to the selected job title
    $('#jobSuggestions').empty(); // Clear the job suggestions list once a job is selected

    // Fetch skills for the selected job
    $.ajax({
        url: '/get_job_skills', // Endpoint to fetch job-specific skills from the backend
        type: 'GET', // HTTP method used for the request
        data: {job_title: selectedJob}, // Send the selected job title as a parameter in the request
        success: function(response) { // Callback function to handle a successful server response
            var skillsList = $('#skillsList'); // Get the element where the skills will be displayed
            skillsList.empty(); // Clear any existing skills in the list
            if (response.skills && response.skills.length > 0) { // Check if the response contains skills and the length is greater than 0
                response.skills.forEach(function(skill) { // Loops through each skill in the response.skills array
                    skillsList.append('<li><input type="checkbox" name="skills" value="' + skill + '" checked> ' + skill + '</li>');
                    // Creates a checkbox for each skill, marks it as checked, and displays the skill name
                });
            } else {
                skillsList.append('<li>No skills found for this job</li>'); // If no skills are found, display a message to the user
            }
            console.log("Job skills fetched:", response);  // Debugging statement to check the fetched response in the console
        },
        error: function() { // Callback function to handle an error in the request
            alert('Error fetching job skills.'); // Show an alert if the request fails
            console.error("Error fetching job skills");  // Debugging print
        }
    });
});

    // Handle CV upload form submission
    $('#cvUploadForm').on('submit', function(event) { // Add an event listener for the CV upload form submission
        event.preventDefault(); // Prevent the default form submission behavior
        var formData = new FormData(this); // Create a FormData object from the form's data
        $.ajax({
```

```
        url: '/upload_cv', // Endpoint to handle CV uploads on the backend
        type: 'POST', // HTTP method used for the request
        data: formData, // Send the form data to the server
        contentType: false, // Prevents jQuery from setting the content type
        processData: false, // Prevent jQuery from processing the data
        success: function(response) { // Callback function to handle a successful server response
          console.log(response);  // Add this to check the response object
          if (response.error) { // If the server returns an error in the response
            alert(response.error); // Display the error message to the user
            return; // Exit the function if there's an error
          }
          $('#email').val(response.contact_info.email); // Set the email input field with the extracted email from the CV
          $('#phone').val(response.contact_info.phone); // Set the phone input field with the extracted phone number from the
CV
          $('#skillsList').empty(); // Clear any existing skills in the list
          response.skills.forEach(function(skill) { // Loop through the extracted skills in the response
            $('#skillsList').append('<li><input type="checkbox" name="skills" value="' + skill + '" checked> ' + skill + '</li>');
          });
          $('#cv_text').val(response.cv_text); // Store the extracted CV text in a hidden input field
          $('#pdfViewer').attr('src', response.pdf_url); // Sets the source of the iframe to the uploaded PDF URL
          $('#skillsSection').show(); // Show the skills section after a successful upload
          $('#cvUploadForm').hide(); // Hide the CV upload form once the upload is successful
        },
        error: function() { // Callback function to handle an error in the request
          alert('Error uploading CV. Please try again.'); // Display an alert if the CV upload fails
        }
      });
    });

    // Handle skills form submission
    $('#skillsForm').on('submit', function(event) {
      event.preventDefault(); // Prevents the default form submission behavior
      var contactInfoData = $('#contactInfoForm').serialize(); // Serialize contact info form data into a query string format
      var skillsData = $(this).serialize(); // Serialize skills form data into a query string format
      var allData = contactInfoData + '&' + skillsData; // Combine contact info and skills data into a single query string

      $.ajax({
        url: '/process_skills', // Specifies the endpoint where the skills data will be sent
        type: 'POST', // Sends the data using the HTTP POST method
        data: allData, // Passes the serialized skills data to the server
        success: function() { // Callback function that executes if the request succeeds
          alert('Your CV, skills, and experience have been saved!'); // Shows a success alert if the request is successful
          $('#skillsSection').hide(); // Hide the skills section after successful submission
          $('#emptyPageSection').show(); // Show the section containing games/quizzes related to the CV
          loadGames(); // Load available Survys/quizzes based on the submitted CV and skills
        },
        error: function() { // Callback function that executes if the request fails
          alert('Error saving skills and experience. Please try again.'); // Shows an error alert if the request fails
        }
      });
    });

    function loadGames() { // Function to load games data from the server
      $.ajax({ // Start an AJAX request using jQuery
        url: '/games', // URL endpoint to fetch game data
        type: 'GET', // HTTP request type (GET to retrieve data)
        success: function(response) { // Callback function executed on successful response
          console.log(response);  // Add this to check the response object
          const games = response.games; // Extract the 'games' array from the response
          const userSkills = response.user_skills.map(skill => skill.toLowerCase());

          // Ensure the games container is cleared before rendering
          $('#gamesContainer').empty();
```

```
            games.forEach(function(game) {  // Loop through each game object in the 'games' array
              const gameBox = document.createElement('div'); // Create a new <div> element for the game box
              gameBox.style.margin = '20px'; // Add margin around the game box
              gameBox.style.border = '1px solid black'; // Add a black border around the game box
              gameBox.style.padding = '10px'; // Add padding inside the game box
              gameBox.style.width = '150px'; // Set a fixed width for the game box
              gameBox.classList.add('game-box'); // Add a CSS class for additional styling

              const gameImg = document.createElement('img'); // Create an <img> element for the game image
              gameImg.src = game.image; // Set the image source to the game's image URL
              gameImg.alt = game.name; // Set the alternative text for the image for accessibility
              gameImg.style.width = '100%'; // Ensure the image scales to fit the width of the box

              const gameTitle = document.createElement('h3'); // Create an <h3> element for the game title
              gameTitle.innerText = game.name; // Set the text of the game title to the game name

              const gameStatus = document.createElement('p'); // Create a <p> element for the game status
              gameStatus.innerText = game.status; // Set the text of the game status to the game's status

              gameBox.appendChild(gameImg); // Append the game image to the game box
              gameBox.appendChild(gameTitle); // Append the game title to the game box
              gameBox.appendChild(gameStatus); // Append the game status to the game box

              gameBox.addEventListener('click', function() { // Add a click event listener to the game box
                console.log("Loading details for:", game.name);  // Debugging to check if the function is triggered
                loadGameDetails(game.name); // Call the loadGameDetails function when the game box is clicked
              });

              document.getElementById('gamesContainer').appendChild(gameBox); // Append the gameBox to the games
container in the DOM
            });
          },
          error: function() { // Callback function executed if the AJAX request fails
            alert('Error loading games. Please try again.'); // Display an alert message if there's an error loading the data
          }
        });
    }

    function loadGameDetails(gameName) { // Send an AJAX request to get the details of a specific game
      $.ajax({
        url: `/game/${gameName}` , // API endpoint to get game details based on gameName
        type: 'GET', // HTTP request type (GET to retrieve data)
        success: function(response) {
          console.log(response);  // Add this to check the response object
          // Check if the response contains an error
          if (response.error) {
            alert(response.error); // Display the error message in an alert
            return; // Exit the function if there's an error
          }
          $('#gameName').text(gameName);  // Sets the name of the game
          $('#gameDescription').text(response.description);  // Sets the description of the game
          $('#gameSkills').text(response.skills_required.join(', '));  // Displays the required skills for the game
          $('#gameImage').attr('src', response.image);  // Displays the image for the game
          // Show the matched skills

          // If matched skills exist, join them into a string; otherwise, display "None" this is done for both the codes below
          //const matchedSkills = response.matched_skills.length ? response.matched_skills.join(', ') : "None";
          //const missingSkills = response.missing_skills.length ? response.missing_skills.join(', ') : "None";
          const matchedSkills = (response.matched_skills && response.matched_skills.length) ?
response.matched_skills.join(', ') : "None";

          const missingSkills = (response.missing_skills && response.missing_skills.length) ? response.missing_skills.join(', ') :
"None";
```

```
            $('#gameMatchedSkills').text("Skills you have: " + matchedSkills);  // Display matched skills
            $('#gameMissingSkills').text("Skills you're missing: " + missingSkills);  // Display missing skills


            $('#gameMatchedSkills').text("Skills you have: " + matchedSkills);  // Display matched skills
            $('#gameMissingSkills').text("Skills you're missing: " + missingSkills);  // Display missing skills
            $('#emptyPageSection').hide();  // Hide the games list
            $('#gameDetailSection').show();  // Show the selected game's details
         },
         error: function() {
            alert('Error loading game details.'); // Show an alert if the request fails
         }
      });
   }
   // Add a click event handler to the "Back to Games" button
   $('#backToGames').on('click', function() {
      $('#gameDetailSection').hide();  // Hide game details
      $('#emptyPageSection').show();  // Show the games list again
   });
</script>
</body>
</html>
```

Appendix A.3 – Phase 2: JD Matching Scoring Function for CVs in database Backend

```
import re # Import regular expressions for text processing
import fitz  # PyMuPDF for PDF handling
import docx  # for handling DOCX files
import os # Provides a way of using operating system dependent functionality
from fuzzywuzzy import fuzz, process  # for fuzzy matching
from pymongo import MongoClient # MongoDB client for database operations
from flask import Flask, render_template, request, jsonify # Flask web framework functions
import os # Duplicate import of os module

app = Flask(__name__) # Initialize the Flask application

# Reuse the same MongoDB
client = MongoClient('mongodb://localhost:27017/')
db = client['cv_analysis2_db']    # same DB name as in your main code
cv_collection = db['cvs2']       # same collection name as in your main code

# Folder to store uploaded JDs (create if not existing)
JD_UPLOAD_FOLDER = os.path.join(os.getcwd(), 'uploaded_jds')
os.makedirs(JD_UPLOAD_FOLDER, exist_ok=True) # Create folder if it doesn't already exist

# --- Helper functions for text extraction (same logic you used for CVs) ---
def extract_text_from_pdf(file_stream):
   doc = fitz.open(stream=file_stream.read(), filetype="pdf") # Open PDF file
   return "\n".join([page.get_text() for page in doc]) # Extract and join text from all pages

def extract_text_from_docx(file_stream):
   docx_obj = docx.Document(file_stream) # Load DOCX file
   return "\n".join([para.text for para in docx_obj.paragraphs]) # Extract and join text from all paragraphs

def preprocess_text(text):
   return re.sub(r'[^a-zA-Z0-9\s]', '', text).lower() # Remove special characters and convert to lowercase

# --- Skill Weights by JD Role ---
skill_weights_by_role = {
   "software engineer": {"java": 2.0, "javascript": 1.8, "react": 1.7, "spring boot": 1.6, "python": 0.9, "docker": 0.7},
   "data scientist": {"tensorflow": 2.5, "sql": 2.5, "python": 2.0, "tableau": 2.0, "data analysis": 2.0, "aws": 1.6},
   "machine learning": {"tensorflow": 2.5,"python": 2.0,"sql": 1.8, "keras": 1.8, "deep learning": 1.6, "scikit-learn": 1.7, "aws":
1.5},
   "cybersecurity analyst": {"penetration testing": 2.5, "siem": 2.5, "ethical hacking": 2.0, "penetration testing": 2.0, "burp
suite": 1.6, "malware analysis": 1.8},
   "cloud architect": {"aws": 1.8, "azure": 1.8, "terraform": 1.5, "kubernets": 1.5, "docker": 1.5},
   "ai engineer": {"transformers": 2.0, "gcp": 2.0, "huggingface": 1.8, "llm": 1.6,},
   "data engineer": {"aws": 2.5, "python": 2.5,"sql": 2.0, "spark": 1.8, "data pipelines": 1.7, "etl pipelines": 1.6},
}

# --- Route to upload JD and match CVs ---
@app.route('/upload_jd', methods=['POST'])
def upload_jd():
   jd_file = request.files.get('jd_file') # Get uploaded JD file
   jd_type = request.form.get('jd_type', '').lower() # Get job role type from form
   if not jd_file or jd_type not in skill_weights_by_role:
      return jsonify({"error": "Invalid file or JD type selection."}), 400

   jd_path = os.path.join(JD_UPLOAD_FOLDER, jd_file.filename)  # Create JD file path
   jd_file.save(jd_path) # Save uploaded JD file

   extension = jd_file.filename.split('.')[-1].lower() # Extract file extension
   try:
      with open(jd_path, 'rb') as f: # Open JD file in binary mode
         jd_text = extract_text_from_pdf(f) if extension == 'pdf' else extract_text_from_docx(f) # Extract text based on extension
   except Exception as e:
      return jsonify({"error": str(e)}), 500 # Return error if file reading fails

   jd_text_processed = preprocess_text(jd_text) # Clean and preprocess the JD text
```

```
weights = skill_weights_by_role[jd_type] # Get skill weights for the selected job type
# List of possible Technical skills
possible_skills = [
# Programming Languages
"python", "r", "sql", "java", "c++", "c#", "scala", "javascript", "typescript", "go",
"ruby", "php", "kotlin", "swift", "rust", "bash", "powershell", "matlab", "objective-c",
"perl", "dart", "haskell", "lua", "f#", "julia", "groovy", "shell scripting", "smalltalk",
"spark",

# Machine Learning & AI
"tensorflow", "pytorch", "scikit-learn", "keras", "huggingface", "openai", "chatgpt",
"bert", "gpt-3", "llm", "transformers", "xgboost", "lightgbm", "catboost", "nltk",
"spacy", "deep learning", "cnn", "rnn", "gan", "vae", "autoencoders",
"natural language processing", "transfer learning", "fine-tuning", "reinforcement learning",
"graph neural networks", "contrastive learning", "clustering", "classification",
"anomaly detection", "generative ai", "few-shot learning", "self-supervised learning",
"semi-supervised learning", "zero-shot learning", "bayesian networks","data analysis",
"data preprocessing","machine learning", "scikit-learn", "nlp", "neural networks",
"data preprocessing", "statistical modeling", "data mining",
"fraud detection", "ai-driven threat detection", "predictive modeling", "image recognition",
"image classification", "object detection", "sentiment analysis", "collaborative filtering",

# Data Science & Analytics
"pandas", "numpy", "scipy", "matplotlib", "seaborn", "plotly", "dash", "statsmodels",
"shap", "lime", "mlflow", "kubeflow", "feature engineering", "data cleaning",
"data wrangling", "time series analysis", "statistical modeling", "hypothesis testing",
"ab testing", "anova", "decision trees", "random forests", "support vector machines",
"gradient boosting", "dimensionality reduction", "feature extraction", "data pipelines",
"big data management", "data analysis",
"demand forecasting", "etl pipelines","tableau",

# Cloud Platforms & Services
"aws", "azure", "gcp", "ibm cloud", "oracle cloud", "aws lambda", "aws ec2", "aws s3",
"big query", "cloud run", "cloud functions", "vertex ai", "aws glue", "redshift",
"cloudwatch", "azure synapse", "azure databricks", "blob storage", "aws sagemaker",
"aws cloudformation", "terraform","eks", "aks", "gke", "helm",
"elastic beanstalk", "azure app service", "firebase", "cloud storage", "cloud security",
"cloud migration", "hybrid cloud", "multi-cloud", "google cloud",
"google ai platform", "azure ml studio", "cloud computing",

# Web Development
"html5", "css3", "sass", "less", "bootstrap", "tailwind", "javascript", "jquery",
"angular", "vue.js", "next.js", "nuxt.js", "svelte", "astro", "django",
"flask", "fastapi", "express", "nestjs", "hapi.js", "graphql", "apollo",
"socket.io", "rest api", "soap", "grpc",

# Backend Development
"node.js", "spring boot", "express.js", "django", "flask", "fastapi", "rust actix",
"go fiber", "java servlet", "php laravel", "ruby on rails", "gin", "asp.net",

# Frontend Frameworks
"react", "angular", "vue", "svelte", "solidjs", "preact", "next.js", "nuxt.js",
"tailwind", "bootstrap", "material ui", "chakra ui",

# Mobile Development
"react native", "flutter", "kotlin", "swift", "xamarin", "phonegap", "capacitor",
"android studio", "xcode",

# DevOps & Automation
"docker", "kubernetes", "terraform", "ansible", "puppet", "chef", "jenkins", "circleci",
"travis ci", "gitlab ci/cd", "github actions", "argo cd", "vault", "helm", "docker compose",
"spinnaker", "prometheus", "grafana", "splunk", "datadog", "qualys",

# Networking & Security
```

```
    "tcp/ip", "udp", "dns", "vpn", "firewall", "load balancing", "ssl/tls", "https",
    "http/2", "http/3", "ipv4", "ipv6", "proxy", "nginx", "apache", "cisco ios",
    "bgp", "ospf", "eigrp", "wireshark", "pfSense", "ipsec", "openvpn", "cloudflare",

    # Cybersecurity
    "penetration testing", "ethical hacking", "metasploit", "nessus", "burp suite",
    "red team", "blue team", "vulnerability scanning", "threat hunting", "malware analysis",
    "siem", "soc", "snort", "cylance", "crowdstrike", "carbon black", "darktrace",
    "ssl inspection", "sandboxing", "network segmentation",
    "security orchestration", "ai-driven threat detection", "incident response",
    "encryption protocols", "firewall configuration", "ids/ips", "risk assessment",
    "patch management", "endpoint protection", "splunk", "qualys", "wireshark", "cisco",
    "security auditing", "active directory", "group policy", "cybersecurity frameworks (iso 27001)",
    "iso 27001", "cybersecurity frameworks (nist)", "cis controls)", "firewalls", "malware detection",
    "owasp top 10", "threat mitigation", "vulnerability management",

    # Database Management
    "mysql", "postgresql", "mongodb", "redis", "sqlite", "oracle db", "ms sql server",
    "couchbase", "neo4j", "elasticsearch", "dynamodb", "amazon aurora", "firestore",
    "influxdb", "timescaledb",

    # Testing & QA
    "selenium", "cypress", "playwright", "junit", "pytest", "testng", "mocha",
    "jasmine", "karma", "appium", "loadrunner", "postman", "rest assured",

    # Operating Systems
    "linux", "ubuntu", "debian", "red hat", "centos", "kali linux", "windows server",
    "macos", "unix", "freebsd", "openbsd",

    # Blockchain & Cryptography
    "ethereum", "solidity", "bitcoin", "hyperledger", "cardano", "defi", "zero knowledge proofs",
    "cryptography", "openssl", "jwt", "sha256", "elliptic curve",

    # Data Formats & Parsing
    "json", "xml", "csv", "parquet", "orc", "yaml", "protobuf", "avro",

    # Search & Indexing
    "elasticsearch", "solr", "opensearch", "algolia", "redisearch", "vector search",
    "k-nearest neighbors"
]
    # Normalize all skills in the list by stripping whitespace and converting to lowercase
    possible_skills = [skill.strip().lower() for skill in possible_skills]

    jd_skills_found = [] # Initialize an empty list to store matched skills found in the JD
    # Use fuzzy matching to find the top 20 most similar skills from possible_skills that match the JD text
    potential_matches = process.extract(jd_text_processed, possible_skills, scorer=fuzz.token_set_ratio, limit=20)

    # Loop through each potential match and its associated fuzzy score
    for match, score in potential_matches:
      threshold = 65 if ' ' in match else 70 # Set a lower threshold for multi-word terms 65 and higher for single-word terms 70
      if score > threshold: # Only consider matches that exceed the threshold
        if len(match) <= len(jd_text_processed) + 10: # Basic length penalty check to avoid matches that are too long
          # First context check: Ensure full word match or high partial ratio to avoid false positives
          if f" {match} " in f" {jd_text_processed} " or fuzz.partial_ratio(match, jd_text_processed) > 80:
            jd_skills_found.append(match)
          # Second context check: Check each word in the multi-word phrase for high similarity
          elif any(fuzz.partial_ratio(word.lower(), jd_text_processed.lower()) > 80 for word in match.split()):
            jd_skills_found.append(match)

    all_cvs = list(cv_collection.find({})) # Retrieve all CVs from MongoDB
    cv_matches = [] # List to store matching CVs and scores
    for cv in all_cvs:
      cv_skills_lower = [s.lower() for s in cv.get("skills", [])] # Normalize CV skill list
      matched_skills = set(jd_skills_found).intersection(set(cv_skills_lower)) # Find common skills
```

```
        weighted_score = sum(weights.get(skill, 1) for skill in matched_skills) # Calculate weighted score
        max_possible = sum(weights.get(skill, 1) for skill in jd_skills_found) # Max possible score
        score = (weighted_score / max_possible) * 100 if max_possible else 0 # Final score as percentage

        cv_matches.append({ # Append matched CV details
            "cv_id": str(cv.get("_id")),
            "contact_info": cv.get("contact_info", {}),
            "education": cv.get("education", ""),
            "experience": cv.get("experience", ""),
            "skills": cv.get("skills", []),
            "years_of_experience": cv.get("years_of_experience", ""),
            "filename": cv.get("filename", ""),
            "matched_skills": list(matched_skills),
            "match_score": score
        })

    cv_matches_sorted = sorted(cv_matches, key=lambda x: x["match_score"], reverse=True) # Sort by match score
    return jsonify({"jd_skills_found": jd_skills_found, "cv_matches": cv_matches_sorted}) # Return matches as JSON

@app.route('/') # Root route
def index():
    return render_template('ModifiedLSBUPhase2.html') # Load front-end HTML page

if __name__ == "__main__": # Run app if script is executed directly
    app.run(debug=True, port=5002) # Start Flask server on port 5002
```

Appendix A.4 – Phase 2: JD Matching Scoring Function for CVs in database Frontend

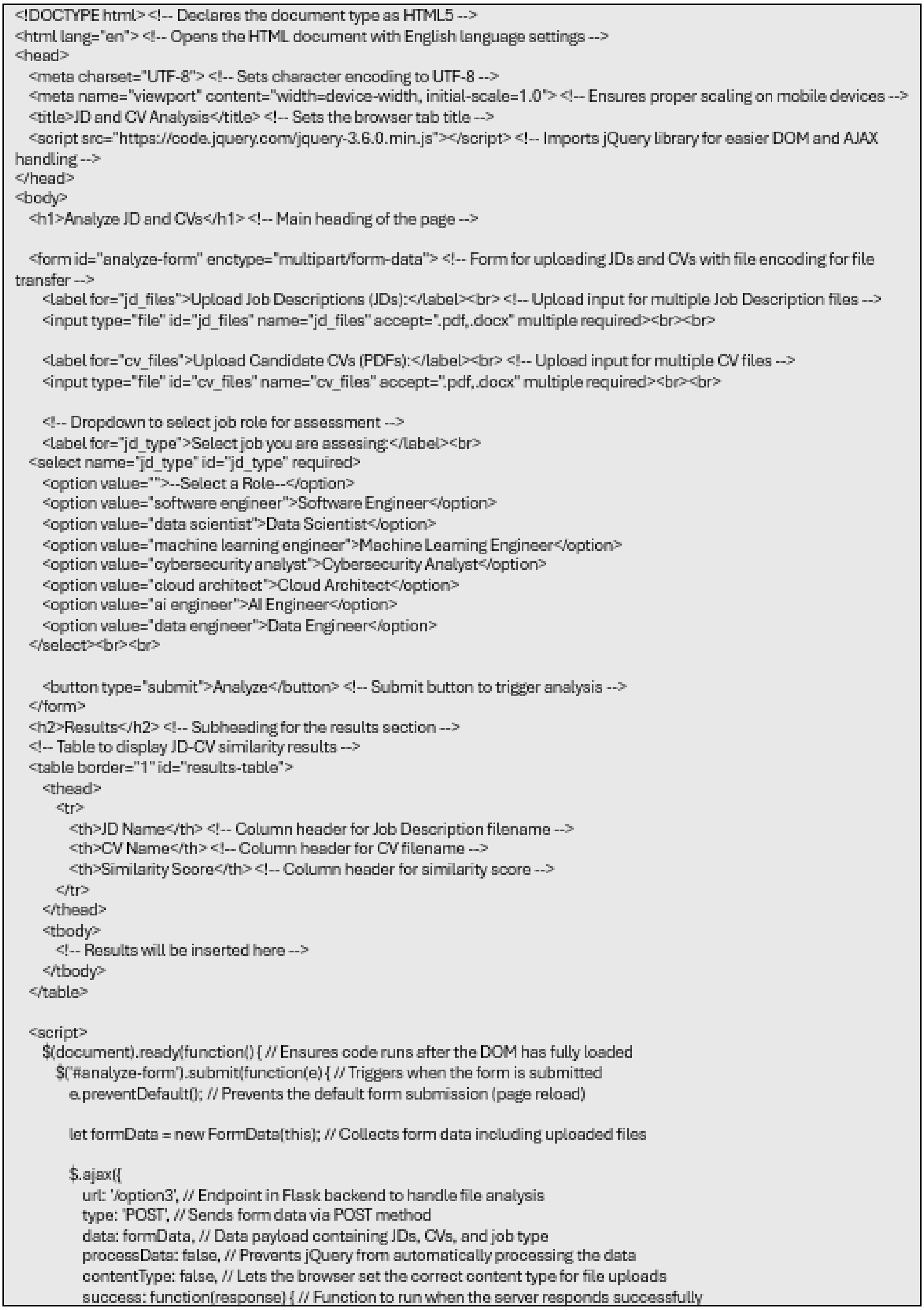
```
<!DOCTYPE html> <!-- Defines the document type as HTML5 -->
<html lang="en"> <!-- Opens the HTML document and sets the language to English -->
<head> <!-- Head section for metadata and scripts -->
 <meta charset="UTF-8"> <!-- Sets character encoding to UTF-8 -->
 <title>JD Matching</title> <!-- Title shown in browser tab -->
 <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script> <!-- Imports jQuery library -->
</head>
<body> <!-- Opens the body of the HTML document -->
 <h1>Upload Your Job Description to Find Matching CVs</h1> <!-- Heading for the page -->
 <form id="jdUploadForm" method="POST" enctype="multipart/form-data"> <!-- Form to upload JD file and select type -->
  <label for="jd_file">Select JD File (PDF or DOCX):</label> <!-- Label for JD file input -->
  <input type="file" id="jd_file" name="jd_file" required><br><br> <!-- File input field for JD upload -->

  <label for="jd_type">Select JD Type:</label> <!-- Label for job type dropdown -->
  <select name="jd_type" id="jd_type" required> <!-- Dropdown to select JD role -->
   <option value="">--Select a Role--</option> <!-- Placeholder option -->
   <option value="software engineer">Software Engineer</option> <!-- Job option -->
   <option value="data scientist">Data Scientist</option> <!-- Job option -->
   <option value="machine learning">Machine Learning</option> <!-- Job option -->
   <option value="cybersecurity analyst">Cybersecurity Analyst</option> <!-- Job option -->
   <option value="cloud architect">Cloud Architect</option> <!-- Job option -->
   <option value="ai engineer">AI Engineer</option> <!-- Job option -->
   <option value="data engineer">Data Engineer</option> <!-- Job option -->
  </select><br><br> <!-- Line break after dropdown -->

  <button type="submit">Upload JD</button> <!-- Submit button to send the form -->
 </form>

 <div id="results" style="display: none;"> <!-- Results section, initially hidden -->
  <h2>Job Description Skills Found</h2> <!-- Subheading for found skills -->
  <ul id="jdSkillsList"></ul> <!-- Unordered list to display matched JD skills -->

  <h2>Matching CVs</h2> <!-- Subheading for matching CVs -->
  <table border="1" id="cvMatchesTable"> <!-- Table to show matched CVs -->
   <thead> <!-- Table header section -->
    <tr> <!-- Table row for header labels -->
     <th>CV ID</th> <!-- Column for CV ID -->
     <th>Match Score (%)</th> <!-- Column for match score -->
     <th>Matched Skills</th> <!-- Column for matched skills -->
     <th>CV Skills</th> <!-- Column for all CV skills -->
     <th>Contact Info</th> <!-- Column for contact details -->
    </tr>
   </thead>
   <tbody> <!-- Table body for CV rows -->
   <!-- Rows dynamically populated -->
   </tbody>
  </table>
 </div>

<script>
$(document).ready(function(){ // Wait until the document is fully loaded
 $('#jdUploadForm').on('submit', function(e){ // Event listener for form submission
  e.preventDefault(); // Prevent the default form submission
  var formData = new FormData(this); // Create FormData object from form inputs

  $.ajax({ // Start AJAX request to server
   url: '/upload_jd', // Endpoint to send data to
   type: 'POST', // HTTP method
   data: formData, // Data being sent
   contentType: false, // Let browser set content type automatically
   processData: false, // Prevent jQuery from processing data
   success: function(response){ // If the request is successful
    if(response.error){ // Check if server returned an error
     alert(response.error); // Show error alert
```

```
          return; // Stop further execution
        }

        $('#results').show();// Show results

        $('#jdSkillsList').empty(); // Clear previous skill results
        response.jd_skills_found.forEach(function(skill){ // Loop through found skills
          $('#jdSkillsList').append('<li>' + skill + '</li>'); // Add skill to the list
        });

        // Populate CV matches table
        var tbody = $('#cvMatchesTable tbody'); // Select table body element
        tbody.empty(); // Clear previous CV match rows
        response.cv_matches.forEach(function(cv){ // Loop through matched CVs
          var matchedSkillsString = cv.matched_skills.join(', '); // Convert matched skills to string
          var allSkillsString = cv.skills.join(', '); // Convert all CV skills to string
          var email = cv.contact_info.email ? cv.contact_info.email : "N/A"; // Get email or show N/A
          var phone = cv.contact_info.phone ? cv.contact_info.phone : "N/A"; // Get phone or show N/A

          var row = '<tr>' + // Build table row with CV data
                '<td>' + cv.cv_id + '</td>' +
                '<td>' + cv.match_score.toFixed(2) + '</td>' +
                '<td>' + matchedSkillsString + '</td>' +
                '<td>' + allSkillsString + '</td>' +
                '<td>Email: ' + email + '<br>Phone: ' + phone + '</td>' +
               '</tr>';
          tbody.append(row); // Append the row to the table
        });
      },
      error: function(){ // If AJAX request fails
        alert('Error uploading JD. Please try again.'); // Show error alert
      }
    });
  });
});
</script>
</body>
</html>
```

Appendix A.5 – Phase 3 CV and JD scoring Backend

```
#This code Should work with the "LSBUPhase3.html template they everything worked last check was on 27/01/2025"
from flask import Flask, render_template, request, jsonify # Import necessary Flask modules for building the web application
from sentence_transformers import SentenceTransformer, util
import fitz  # PyMuPDF for PDF handling and text extraction from PDF
import os # Import os for setting environment variables and handling file system operations
from fuzzywuzzy import fuzz, process  # for fuzzy matching
import docx # For extracting text from DOCX files
import re # Regular expressions for text preprocessing

os.environ['TF_ENABLE_ONEDNN_OPTS'] = '0' # Suppress TensorFlow warnings
app = Flask(__name__, template_folder='templates') # Initialize Flask app and specify the folder for HTML templates
model = SentenceTransformer('all-mpnet-base-v2') # Load Sentence Transformer model  paraphrase-MiniLM-L6-v2, all-
mpnet-base-v2

# Full universal skill list for fuzzy matching (add more as needed)
ALL_POSSIBLE_SKILLS = [
    # Programming Languages
    "python", "r", "sql", "java", "c++", "c#", "scala", "javascript", "typescript", "go",
    "ruby", "php", "kotlin", "swift", "rust", "bash", "powershell", "matlab", "objective-c",
    "perl", "dart", "haskell", "lua", "f#", "julia", "groovy", "shell scripting", "smalltalk",
    "spark",

    # Machine Learning & AI
    "tensorflow", "pytorch", "scikit-learn", "keras", "huggingface", "openai", "chatgpt",
    "bert", "gpt-3", "llm", "transformers", "xgboost", "lightgbm", "catboost", "nltk",
    "spacy", "deep learning", "cnn", "rnn", "gan", "vae", "autoencoders",
    "natural language processing", "transfer learning", "fine-tuning", "reinforcement learning",
    "graph neural networks", "contrastive learning", "clustering", "classification",
    "anomaly detection", "generative ai", "few-shot learning", "self-supervised learning",
    "semi-supervised learning", "zero-shot learning", "bayesian networks","data analysis",
    "data preprocessing","machine learning", "scikit-learn", "nlp", "neural networks",
    "data preprocessing", "statistical modeling", "data mining",
    "fraud detection", "ai-driven threat detection", "predictive modeling", "image recognition",
    "image classification", "object detection", "sentiment analysis", "collaborative filtering",

    # Data Science & Analytics
    "pandas", "numpy", "scipy", "matplotlib", "seaborn", "plotly", "dash", "statsmodels",
    "shap", "lime", "mlflow", "kubeflow", "feature engineering", "data cleaning",
    "data wrangling", "time series analysis", "statistical modeling", "hypothesis testing",
    "ab testing", "anova", "decision trees", "random forests", "support vector machines",
    "gradient boosting", "dimensionality reduction", "feature extraction", "data pipelines",
    "big data management", "data analysis",
    "demand forecasting", "etl pipelines","tableau",

    # Cloud Platforms & Services
    "aws", "azure", "gcp", "ibm cloud", "oracle cloud", "aws lambda", "aws ec2", "aws s3",
    "big query", "cloud run", "cloud functions", "vertex ai", "aws glue", "redshift",
    "cloudwatch", "azure synapse", "azure databricks", "blob storage", "aws sagemaker",
    "aws cloudformation", "terraform","eks", "aks", "gke", "helm",
    "elastic beanstalk", "azure app service", "firebase", "cloud storage",
    "cloud migration", "hybrid cloud", "multi-cloud",
    "google ai platform", "azure ml studio",

    # Web Development
    "html5", "css3", "sass", "less", "bootstrap", "tailwind", "javascript", "jquery",
    "angular", "vue.js", "next.js", "nuxt.js", "svelte", "astro", "django",
    "flask", "fastapi", "express", "nestjs", "hapi.js", "graphql", "apollo",
    "socket.io", "rest api", "soap", "grpc",

    # Backend Development
    "node.js", "spring boot", "express.js", "django", "flask", "fastapi", "rust actix",
    "go fiber", "java servlet", "php laravel", "ruby on rails", "gin", "asp.net",

    # Frontend Frameworks
    "react", "angular", "vue", "svelte", "solidjs", "preact", "next.js", "nuxt.js",
    "tailwind", "bootstrap", "material ui", "chakra ui",

    # Mobile Development
    "react native", "flutter", "kotlin", "swift", "xamarin", "phonegap", "capacitor",
    "android studio", "xcode",

    # DevOps & Automation
    "docker", "kubernetes", "terraform", "ansible", "puppet", "chef", "jenkins", "circleci",
    "travis ci", "gitlab ci/cd", "github actions", "argo cd", "vault", "helm", "docker compose",
    "spinnaker", "prometheus", "grafana", "splunk", "datadog", "qualys",

    # Networking & Security
    "tcp/ip", "udp", "dns", "vpn", "firewall", "load balancing", "ssl/tls", "https",
    "http/2", "http/3", "ipv4", "ipv6", "proxy", "nginx", "apache", "cisco ios",
    "bgp", "ospf", "eigrp", "wireshark", "pfSense", "ipsec", "openvpn", "cloudflare",

    # Cybersecurity
    "penetration testing", "ethical hacking", "metasploit", "nessus", "burp suite",
    "red team", "blue team", "vulnerability scanning", "threat hunting", "malware analysis",
    "siem", "soc", "snort", "cylance", "crowdstrike", "carbon black", "darktrace",
    "ssl inspection", "sandboxing", "network segmentation",
    "security orchestration", "ai-driven threat detection", "incident response",
    "encryption protocols", "firewall configuration", "ids/ips", "risk assessment",
    "patch management", "endpoint protection", "splunk", "qualys", "wireshark", "cisco",
    "security auditing", "active directory", "group policy", "cybersecurity frameworks (iso 27001)",
    "iso 27001", "cybersecurity frameworks (nist)", "cis controls)", "firewalls", "malware detection",
    "owasp top 10", "threat mitigation", "vulnerability management",

    # Database Management
    "mysql", "postgresql", "mongodb", "redis", "sqlite", "oracle db", "ms sql server",
    "couchbase", "neo4j", "elasticsearch", "dynamodb", "amazon aurora", "firestore",
    "influxdb", "timescaledb",

    # Testing & QA
    "selenium", "cypress", "playwright", "junit", "pytest", "testng", "mocha",
    "jasmine", "karma", "appium", "loadrunner", "postman", "rest assured",

    # Operating Systems
    "linux", "ubuntu", "debian", "red hat", "centos", "kali linux", "windows server",
    "macos", "unix", "freebsd", "openbsd",

    # Blockchain & Cryptography
    "ethereum", "solidity", "bitcoin", "hyperledger", "cardano", "defi", "zero knowledge proofs",
    "cryptography", "openssl", "jwt", "sha256", "elliptic curve",

    # Data Formats & Parsing
    "json", "xml", "csv", "parquet", "orc", "yaml", "protobuf", "avro",

    # Search & Indexing
    "elasticsearch", "solr", "opensearch", "algolia", "redisearch", "vector search",
    "k-nearest neighbors"
]

ALL_POSSIBLE_SKILLS = list(set(skill.strip().lower() for skill in ALL_POSSIBLE_SKILLS))

# Predefined skill weights by JD type (now extended)
skill_weights_by_jd = {
    "software engineer": {"java": 2.0, "javascript": 1.8, "react": 1.7, "spring boot": 1.6, "python": 0.9, "docker": 0.7},
    "data scientist": {"tensorflow": 2.5, "sql": 2.5, "python": 2.0, "tableau": 2.0, "data analysis": 2.0, "aws": 1.6},
    "machine learning": {"tensorflow": 2.5,"python": 2.0,"sql": 1.8, "keras": 1.8, "deep learning": 1.6, "scikit-learn": 1.7, "aws":
1.5},
    "cybersecurity analyst": {"penetration testing": 2.5, "siem": 2.5, "ethical hacking": 2.0, "penetration testing": 2.0, "burp
suite": 1.6, "malware analysis": 1.8},
    "cloud architect": {"azure": 1.8, "terraform": 1.5, "docker": 1.5},
    "ai engineer": {"transformers": 1.1, "huggingface": 1.8, "llm": 1.6, "python": 1.1},
    "data engineer": {"aws": 2.5, "python": 2.5,"sql": 2.0, "spark": 1.8, "data pipelines": 1.7, "etl pipelines": 1.6},
}

# Function to extract text from uploaded PDF or DOCX
def extract_text_from_file(file):
    try:
        filename = file.filename.lower() # Get lowercase filename
        if filename.endswith('.pdf'): # If PDF
            doc = fitz.open(stream=file.read(), filetype="pdf") # Open PDF stream
            return " ".join([page.get_text() for page in doc]) # Join text from all pages
        elif filename.endswith('.docx'): # If DOCX
            doc = docx.Document(file) # Read DOCX file
            return "\n".join([para.text for para in doc.paragraphs]) # Join all paragraph texts
        else:
            raise ValueError("Unsupported file format. Please upload a PDF or DOCX file.") # Raise error for unsupported file
    except Exception as e:
        print(f"Error extracting text: {str(e)}") # Print error message
        raise # Re-raise the exception

# Function to extract skills from text using fuzzy matching
def custom_extract_skills(text, skills_list):
    skills_found = [] # Initialize list of found skills
    text = re.sub(r'\s+', ' ', text).strip().lower() # Normalize whitespace and lowercase
    skills_list = [re.sub(r'\s+', ' ', s).strip().lower() for s in skills_list] # Normalize skills list
    all_skills = list(set(skills_list))  # remove duplicates

    # Extract potential matches using fuzzy token_set_ratio
    potential_matches = process.extract(text, all_skills, scorer=fuzz.token_set_ratio, limit=50)

    for match, score in potential_matches: # Loop through potential matches
        threshold = 65 if ' ' in match else 70 # Lower threshold for multi-word skills
        if score > threshold:
            if len(match) <= len(text) + 10: # Basic length check
                # Context checks to reduce false positives
                if f" {match} " in f" {text} " or fuzz.partial_ratio(match, text) > 80:
                    skills_found.append(match)
                elif any(fuzz.partial_ratio(word.lower(), text.lower()) > 80 for word in match.split()):
                    skills_found.append(match)
    return skills_found # Return matched skills

# Basic text preprocessing (remove special characters and lowercase)
def preprocess_text(text):
    return re.sub(r'[^a-zA-Z0-9\s]', '', text).lower()

# Home route renders HTML page
@app.route('/')
def home():
    return render_template('ModifiedLSBUPhase3.html') # Load the Phase 3 template

# Option 3 route handles file uploads and comparison logic
@app.route('/option3', methods=['GET', 'POST'])
def option3():
    if request.method == 'POST': # Handle POST request
        jd_files = request.files.getlist('jd_files') # Get list of uploaded JD files
        jd_type = request.form.get('jd_type', '').lower() # Get JD type and lowercase it
        cv_files = request.files.getlist('cv_files') # Get list of uploaded CV files

        if not jd_files: # Check if any JD files uploaded
            return jsonify({'error': 'Please upload at least one JD file'}), 400
        if not cv_files: # Check if any CV files uploaded
            return jsonify({'error': 'Please upload at least one CV file'}), 400
        jd_texts = [extract_text_from_file(jd) for jd in jd_files] # Extract text from each JD
        cv_texts = [extract_text_from_file(cv) for cv in cv_files] # Extract text from each CV

        jd_embeddings = [model.encode(jd_text, convert_to_tensor=True) for jd_text in jd_texts] # Embed JD texts
        cv_embeddings = [model.encode(cv_text, convert_to_tensor=True) for cv_text in cv_texts] # Embed CV texts

        results = [] # Initialize list for results
        for jd_idx, jd_embedding in enumerate(jd_embeddings):  # Loop through each JD
            jd_text_processed = preprocess_text(jd_texts[jd_idx]) # Preprocess JD text
            jd_weights = skill_weights_by_jd.get(jd_type, {}) # Get skill weights for JD type
            filtered_possible_skills = list(set(ALL_POSSIBLE_SKILLS).union(set(jd_weights.keys()))) # Merge weights + full skill list

            jd_skills_found = custom_extract_skills(jd_text_processed, filtered_possible_skills) # Extract JD skills

            print(f"\n JD {jd_files[jd_idx].filename} matched skills: {jd_skills_found}") # Debug print
            max_possible = sum(jd_weights.get(skill, 1) for skill in jd_skills_found) # Max weight for JD

            for cv_idx, cv_embedding in enumerate(cv_embeddings): # Loop through each CV
                score = util.pytorch_cos_sim(jd_embedding, cv_embedding).item() # Semantic similarity score
                cv_text_processed = preprocess_text(cv_texts[cv_idx]) # Preprocess CV text
                cv_skills_lower = custom_extract_skills(cv_text_processed, filtered_possible_skills) # Extract CV skills

                matched_skills = set(jd_skills_found).intersection(set(cv_skills_lower)) # Find shared skills
                print(f" CV {cv_files[cv_idx].filename} extracted skills: {cv_skills_lower}") # Debug print
                print(f" Matched Skills between JD and CV: {matched_skills}") # Debug print

                weighted_score = sum(jd_weights.get(skill, 1) for skill in matched_skills) # Calculate weight score
                keyword_component = (weighted_score / max_possible) if max_possible > 0 else 0 # Normalize keyword score
                final_score = round((0.65 * score) + (0.35 * keyword_component), 4) # Final combined score

                results.append({ # Append result to the list
                    "jd_name": jd_files[jd_idx].filename,
                    "cv_name": cv_files[cv_idx].filename,
                    "similarity_score": final_score
                })

        return jsonify({'results': results}) # Return results as JSON
    return render_template('ModifiedLSBUPhase3.html') # Render template for GET request

# Run the app locally on port 5001
if __name__ == "__main__":
    app.run(debug=True, port=5001)
```

Appendix A.6 – Phase 3 CV and JD scoring Frontend



```
<!DOCTYPE html> <!-- Declares the document type as HTML5 -->
<html lang="en"> <!-- Opens the HTML document with English language settings -->
<head>
  <meta charset="UTF-8"> <!-- Sets character encoding to UTF-8 -->
  <meta name="viewport" content="width=device-width, initial-scale=1.0"> <!-- Ensures proper scaling on mobile devices -->
  <title>JD and CV Analysis</title> <!-- Sets the browser tab title -->
  <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script> <!-- Imports jQuery library for easier DOM and AJAX
handling -->
</head>
<body>
  <h1>Analyze JD and CVs</h1> <!-- Main heading of the page -->

  <form id="analyze-form" enctype="multipart/form-data"> <!-- Form for uploading JDs and CVs with file encoding for file
transfer -->
    <label for="jd_files">Upload Job Descriptions (JDs):</label><br> <!-- Upload input for multiple Job Description files -->
    <input type="file" id="jd_files" name="jd_files" accept=".pdf,.docx" multiple required><br><br>

    <label for="cv_files">Upload Candidate CVs (PDFs):</label><br> <!-- Upload input for multiple CV files -->
    <input type="file" id="cv_files" name="cv_files" accept=".pdf,.docx" multiple required><br><br>

    <!-- Dropdown to select job role for assessment -->
    <label for="jd_type">Select job you are assesing:</label><br>
  <select name="jd_type" id="jd_type" required>
    <option value="">--Select a Role--</option>
    <option value="software engineer">Software Engineer</option>
    <option value="data scientist">Data Scientist</option>
    <option value="machine learning engineer">Machine Learning Engineer</option>
    <option value="cybersecurity analyst">Cybersecurity Analyst</option>
    <option value="cloud architect">Cloud Architect</option>
    <option value="ai engineer">AI Engineer</option>
    <option value="data engineer">Data Engineer</option>
  </select><br><br>

    <button type="submit">Analyze</button> <!-- Submit button to trigger analysis -->
  </form>
  <h2>Results</h2> <!-- Subheading for the results section -->
  <!-- Table to display JD-CV similarity results -->
  <table border="1" id="results-table">
    <thead>
      <tr>
        <th>JD Name</th> <!-- Column header for Job Description filename -->
        <th>CV Name</th> <!-- Column header for CV filename -->
        <th>Similarity Score</th> <!-- Column header for similarity score -->
      </tr>
    </thead>
    <tbody>
      <!-- Results will be inserted here -->
    </tbody>
  </table>

  <script>
    $(document).ready(function() { // Ensures code runs after the DOM has fully loaded
      $('#analyze-form').submit(function(e) { // Triggers when the form is submitted
        e.preventDefault(); // Prevents the default form submission (page reload)

        let formData = new FormData(this); // Collects form data including uploaded files

        $.ajax({
          url: '/option3', // Endpoint in Flask backend to handle file analysis
          type: 'POST', // Sends form data via POST method
          data: formData, // Data payload containing JDs, CVs, and job type
          processData: false, // Prevents jQuery from automatically processing the data
          contentType: false, // Lets the browser set the correct content type for file uploads
          success: function(response) { // Function to run when the server responds successfully
```

```
              let results = response.results; // Extracts the 'results' list from the server response
              let resultsTable = $('#results-table tbody'); // Selects the table body where results will be shown
              resultsTable.empty(); // Clears any existing rows in the table

              // Loops through each result and appends it to the table
              results.forEach(result => {
                resultsTable.append(`
                  <tr>
                    <td>${result.jd_name}</td> <!-- JD filename -->
                    <td>${result.cv_name}</td>
                    <td>${result.similarity_score}</td>
                  </tr>
                `);
              });
            },
            error: function(err) { // Function to run if there is an error from the backend
              alert('Error: ' + err.responseJSON.error); // Displays error message to the user
            }
          });
        });
      });
    </script>
  </body>
</html>
```

Appendix B – Additional Heatmaps, graphs, and Testing

Appendix B.1 – Checking how fuzz scoring works to Fine tune Accordingly

```
from fuzzywuzzy import fuzz # Import the fuzzy string matching function to calculate similarity ratios

cv_text = """a1b2c3x9y8z"""  # To store the substring of cv_text that best matches the skill
skill = 'abc123xyz' # Target skill string we want to find within the CV using fuzzy matching

# Initialize tracking variables
max_score = 0 # To keep track of the highest similarity score found so far
best_match = None  # To store the substring of cv_text that best matches the skill

# Sliding window search
for i in range(len(cv_text) - len(skill) + 1): # Loop through cv_text with a moving window based on skill length
  # Create a window around the skill length (+5 to allow for context)
  window = cv_text[i:i + len(skill) + 5] # Extract a substring window from cv_text for comparison

  # Compute fuzzy similarity score
  score = fuzz.partial_ratio(skill.lower(), window.lower())

  if score > max_score: # If this score is higher than the best so far, update best match
    max_score = score
    best_match = window

# Output the best match and score
if max_score > 0: # If a match was found with a positive score
  print(f"Best match: '{best_match}' with score: {max_score}") # Display the substring and its fuzzy match score
else:
  print("No match found.") # No sufficiently similar match was found in the text
```

Appendix B.2 – Generating Heatmaps

```
#This code color codes the heatmap refrencing from the manual scores
# you need to change the file paths here

import pandas as pd  # For reading and processing Excel files
import openpyxl # For Excel file formatting
from openpyxl.styles import PatternFill # For coloring cells in Excel
import os # For checking file existence
import re # For parsing string ranges

# Load Excel files
ground_truth_file = r"C:/Users/zeiad/Python/Vs_code/LSBU/Beng Project/Scores/IT_Sector_Manual_Scores_Percent.xlsx" #
Path to manual scores
model_scores_file = r"C:/Users/zeiad/Python/vs_code/LSBU/Beng Project/Scores/IT_Sector_Models/(1)_all-mpnet-base-
v2/JD_CV_Similarity_Scores_all-mpnet-base-v2.xlsx" # Path to model scores

# Debug file paths
print("Ground Truth File Path:", repr(ground_truth_file))
print("Model Scores File Path:", repr(model_scores_file))

# Check if files exist
if not os.path.exists(ground_truth_file):
  print(f"Error: The ground truth file does not exist at {ground_truth_file}")
else:
  print("Ground truth file exists.") # Confirm the file exists

# Check if ground truth file exists
if not os.path.exists(model_scores_file):
  print(f"Error: The model scores file does not exist at {model_scores_file}")
else:
  print("Model scores file exists.") # Confirm the file exists

# Read files into DataFrames
try:
  ground_truth_df = pd.read_excel(ground_truth_file, index_col=0) # Load with first column as index
  print("Ground truth file loaded successfully.") # Confirm success
except FileNotFoundError:
  print(f"Error: Ground truth file not found at {ground_truth_file}") # Specific file error
  raise # Stop execution
except Exception as e:
  print(f"Error loading ground truth file: {e}") # Other error
  raise # Stop execution

try: # Try loading the model scores Excel file
  model_scores_df = pd.read_excel(model_scores_file, index_col=0) # Load with first column as index
  print("Model scores file loaded successfully.") # Confirm success
except FileNotFoundError:
  print(f"Error: Model scores file not found at {model_scores_file}") # Specific file error
  raise # Stop execution
except Exception as e:
  print(f"Error loading model scores file: {e}")  # Other error
  raise  # Stop execution

# Parse range values in ground truth and convert them to numeric averages
def parse_range(value):
  if isinstance(value, str) and "-" in value: # Check if it's a range string
    match = re.match(r"(\d+)-(\d+)%", value)  # Use regex to extract numbers
    if match:
      lower, upper = map(int, match.groups()) # Convert to integers
      return (lower, upper)  # Return range as tuple
  return None # Return None if not parsable

# Map manual scores to numeric range
ground_truth_df = ground_truth_df.applymap(parse_range) # Convert range strings to tuples
```

```
model_scores_df = model_scores_df.apply(pd.to_numeric, errors='coerce').fillna(0) # Ensure numeric values in model scores

# Ensure DataFrames are aligned
ground_truth_df = ground_truth_df.reindex(index=model_scores_df.index, columns=model_scores_df.columns)

# Load workbook for formatting
wb = openpyxl.Workbook()
ws = wb.active # Get the default worksheet
ws.title = "JD_CV_Heatmap" # Rename the worksheet

# Copy headers
for col_idx, col_name in enumerate(model_scores_df.columns, start=2): # Start from column 2
  ws.cell(row=1, column=col_idx, value=col_name)
for row_idx, row_name in enumerate(model_scores_df.index, start=2): # Start from row 2
  ws.cell(row=row_idx, column=1, value=row_name)

# Define colors for deviation ranges
color_mapping = {
  "dark_green": PatternFill(start_color="32CD32", end_color="32CD32", fill_type="solid"),  # Dark Green (In Range)
  "light_green": PatternFill(start_color="90EE90", end_color="90EE90", fill_type="solid"),  # Light Green (1-5% Out)
  "light_yellow": PatternFill(start_color="FFFF99", end_color="FFFF99", fill_type="solid"),  # 6-10% Out
  "dark_yellow": PatternFill(start_color="FFD700", end_color="FFD700", fill_type="solid"),  # 11-15% Out
  "light_red": PatternFill(start_color="FF6347", end_color="FF6347", fill_type="solid"),  # 16-20% Out
  "dark_red": PatternFill(start_color="8B0000", end_color="8B0000", fill_type="solid"),  # >20% Out
}

# Apply color coding to the cells
for row_idx, (row_name, row_data) in enumerate(model_scores_df.iterrows(), start=2): # Iterate rows
  for col_idx, (col_name, model_value) in enumerate(row_data.items(), start=2): # Iterate columns
    ground_truth_range = ground_truth_df.at[row_name, col_name] # Get the manual score range
    if ground_truth_range is not None and isinstance(ground_truth_range, tuple): # If valid range exists
      lower, upper = ground_truth_range # Unpack range
      if lower <= model_value <= upper:
        fill = color_mapping["dark_green"] # Model score is within manual range
      elif abs(model_value - lower) <= 5 or abs(model_value - upper) <= 5:
        fill = color_mapping["light_green"]  # 1–5% deviation
      elif abs(model_value - lower) <= 10 or abs(model_value - upper) <= 10:
        fill = color_mapping["light_yellow"] # 6–10% deviation
      elif abs(model_value - lower) <= 15 or abs(model_value - upper) <= 15:
        fill = color_mapping["dark_yellow"]  # 11–15% deviation
      elif abs(model_value - lower) <= 20 or abs(model_value - upper) <= 20:
        fill = color_mapping["light_red"] # 16–20% deviation
      else:
        fill = color_mapping["dark_red"] # More than 20% deviation
    else:
      fill = color_mapping["dark_red"]  # Default to dark red if range is missing
    ws.cell(row=row_idx, column=col_idx, value=model_value).fill = fill

# Save the heatmap
wb.save("IT_all-mpnet-base-v2.xlsx") # This will be changed depending on the location
print("Heatmap has been saved to JD_CV_Heatmap.xlsx") # Notify user
```

Appendix B.3 – Generating PCA Semantic Embeddings for CVs and JDs

```
import os # For file path operations and directory listing
import fitz  # PyMuPDF for reading text from PDF files
from sentence_transformers import SentenceTransformer # For generating semantic embeddings
from sklearn.decomposition import PCA # For dimensionality reduction
import matplotlib.pyplot as plt # For plotting
import numpy as np # For numerical operations
from matplotlib.lines import Line2D # For custom legend elements
from adjustText import adjust_text # For automatically adjusting text labels in plots

# --------- CONFIG ---------
cv_dir = "CVS" # Directory containing CV PDF files
jd_dir = "JDS" # Directory containing JD PDF files
model_name = 'sentence-transformers/all-mpnet-base-v2' # Pretrained sentence transformer model

# --------- FUNCTIONS ---------
def extract_text_from_pdf(pdf_path):
   doc = fitz.open(pdf_path) # Open the PDF file
   return " ".join([page.get_text() for page in doc]).strip()  # Join text from all pages

# --------- LOAD TEXTS ---------
cv_texts = [] # List to store extracted CV texts
jd_texts = [] # List to store extracted JD texts
cv_labels = [] # List to store CV filenames
jd_labels = [] # List to store JD filenames

# Loop through all files in the JD directory
for filename in os.listdir(cv_dir):
   if filename.endswith(".pdf"): # Only process PDF files
      text = extract_text_from_pdf(os.path.join(cv_dir, filename)) # Extract text
      cv_texts.append(text) # Append text to CV list
      cv_labels.append(filename) # Append filename to label list

for filename in os.listdir(jd_dir):
   if filename.endswith(".pdf"): # Only process PDF files
      text = extract_text_from_pdf(os.path.join(jd_dir, filename)) # Extract text
      jd_texts.append(text) # Append text to CV list
      jd_labels.append(filename) # Append filename to label list

all_texts = jd_texts + cv_texts # Combine JD and CV texts for embedding

# --------- EMBEDDING ---------
model = SentenceTransformer(model_name) # Load sentence transformer model
embeddings = model.encode(all_texts, normalize_embeddings=True) # Generate and normalize embeddings

# --------- PCA ---------
pca = PCA(n_components=2) # Initialize PCA with 2 components
reduced = pca.fit_transform(embeddings) # Reduce embedding dimensions to 2D

# --------- PLOT ---------
plt.figure(figsize=(12, 8)) # Create a new plot with specified size

texts = [] # List to hold text labels for adjustment

# Plot and label JDs
for i in range(len(jd_texts)):
   x, y = reduced[i] # Get PCA coordinates
   plt.scatter(x, y, c='blue', marker='^', s=100) # Plot JD point as blue triangle
   texts.append(plt.text(x, y, f"JD{i+1}", fontsize=9)) # Add label

# Plot and label CVs
for i in range(len(cv_texts)):
   x, y = reduced[len(jd_texts) + i] # Offset index to get CV points
   plt.scatter(x, y, c='red', marker='o', s=60) # Plot CV point as red circle
   texts.append(plt.text(x, y, f"CV{i+1}", fontsize=9)) # Add label
```

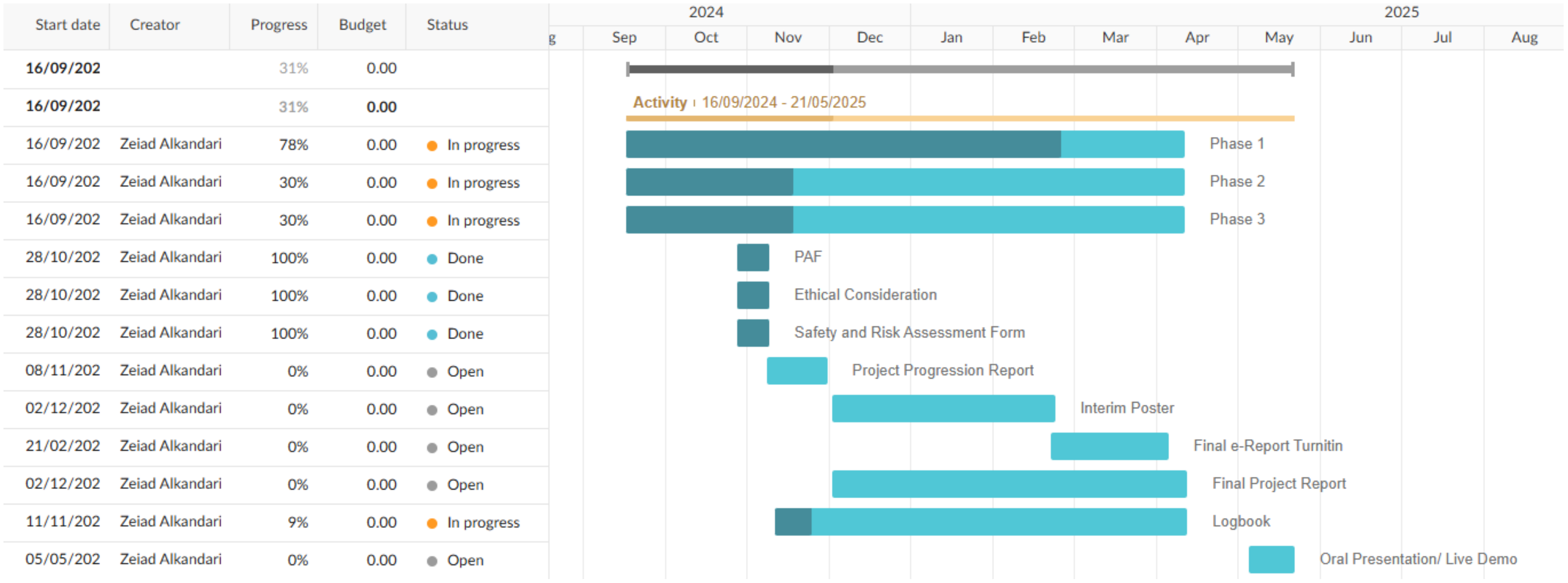
```
# Auto-adjust to avoid label overlaps
adjust_text(
   texts, # List of text objects
   only_move={'points': 'y', 'texts': 'xy'}, # Restrict movement direction
   arrowprops=dict(arrowstyle="-", color='gray', lw=0.5) # Arrow style for displaced labels
)

# Legend for shapes/colors only
legend_elements = [
   Line2D([0], [0], marker='^', color='w', label='Job Descriptions (JDs)', markerfacecolor='blue', markersize=10),
   Line2D([0], [0], marker='o', color='w', label='CVs', markerfacecolor='red', markersize=8)
]
plt.legend(handles=legend_elements, loc='upper right') # Display legend in top-right

plt.title("PCA Visualization of CV and JD Embeddings") # Display legend in top-right
plt.xlabel("PCA Dimension 1") # Label X-axis
plt.ylabel("PCA Dimension 2") # Label Y-axis
plt.grid(True) # Enable grid lines
plt.tight_layout() # Adjust layout to fit everything nicely
plt.savefig("semantic_plot.png", dpi=300) # Save plot as high-res PNG
plt.show() # Display the plot

# --------- PRINT LEGEND ---------
print("\n JD Legend:")
for i, filename in enumerate(jd_labels):
   print(f"JD{i+1}: {filename}")

print("\n CV Legend:") # Print filename legend for CVs
for i, filename in enumerate(cv_labels):
   print(f"CV{i+1}: {filename}")
```

Appendix C – Project Management and Supervision

Appendix C.1 – Meeting with Academic Supervisor

Count	Date	Supervisor
1	**30/10/2025**	**Gokul Thangavel**
2	**01/11/2025**	**Sandra Dudley-Mcevoy**
3	**07/11/2025**	**Sandra Dudley-Mcevoy**
4	**08/11/2025**	**Gokul Thangavel**
5	**07/02/2025**	**Sandra Dudley-Mcevoy**
6	**13/02/2025**	**Sandra Dudley-Mcevoy**
7	**19/03/2025**	**Sandra Dudley-Mcevoy**
8	**24/03/2025**	**Sandra Dudley-Mcevoy**

Appendix C.2 – Project Gannt chart



Start date	Creator	Progress	Budget	Status
16/09/202		31%	0.00	
16/09/202		31%	**0.00**	
16/09/202	Zeiad Alkandari	78%	0.00	● In progress
16/09/202	Zeiad Alkandari	30%	0.00	● In progress
16/09/202	Zeiad Alkandari	30%	0.00	● In progress
28/10/202	Zeiad Alkandari	100%	0.00	● Done
28/10/202	Zeiad Alkandari	100%	0.00	● Done
28/10/202	Zeiad Alkandari	100%	0.00	● Done
08/11/202	Zeiad Alkandari	0%	0.00	● Open
02/12/202	Zeiad Alkandari	0%	0.00	● Open
21/02/202	Zeiad Alkandari	0%	0.00	● Open
02/12/202	Zeiad Alkandari	0%	0.00	● Open
11/11/202	Zeiad Alkandari	9%	0.00	● In progress
05/05/202	Zeiad Alkandari	0%	0.00	● Open