```
# --- Cell 1: Install Dependencies ---
# We add PyPDF2 for parsing PDF resumes and openpyxl for Excel files
!pip install pandas numpy nltk scikit-learn PyPDF2 openpyxl

# --- Cell 2: Import Libraries ---
import pandas as pd
import numpy as np
import re
import string
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from ipywidgets import widgets, Layout, HBox, VBox # For UI
from IPython.display import display, clear_output
import io # For file handling
from PyPDF2 import PdfReader # For PDF parsing
import random # For salary estimation
from collections import Counter

# --- NEW: ML Model Imports ---
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import OneHotEncoder

# --- Cell 3: Download NLTK Data ---
# This is necessary for text preprocessing
print("Downloading NLTK data...")
nltk.download('stopwords')
nltk.download('punkt')
nltk.download('punkt_tab')
print("Downloads complete.")


# --- Cell 4: Text Preprocessing & Info Extraction Functions ---
print("\nDefining text cleaning and extraction functions...")
stop_words = set(stopwords.words('english'))

def clean_resume(text):
    # 1. Convert to lowercase
    text = text.lower()
    # 2. Remove URLs (using raw strings)
    text = re.sub(r'http\S+\s*', ' ', text)
    # 3. Remove RT and cc
```

```
    text = re.sub(r'RT|cc', ' ', text)
    # 4. Remove hashtags (using raw strings)
    text = re.sub(r'#\S+', '', text)
    # 5. Remove mentions (using raw strings)
    text = re.sub(r'@\S+', '  ', text)
    # 6. Remove punctuation
    text = re.sub(r'[%s]' % re.escape(string.punctuation), ' ', text)
    # 7. Remove non-ASCII characters
    text = re.sub(r'[^\x00-\x7f]', r' ', text)
    # 8. Remove extra whitespace (using raw strings)
    text = re.sub(r'\s+', ' ', text)
    # 9. Tokenize and remove stopwords
    words = word_tokenize(text)
    words = [w for w in words if w not in stop_words and len(w) > 2]
    return ' '.join(words)

def parse_pdf(file_content):
    """Extracts text from PDF file content."""
    try:
        pdf_file = io.BytesIO(file_content)
        reader = PdfReader(pdf_file)
        text = ""
        for page in reader.pages:
            text += page.extract_text()
        return text
    except Exception as e:
        return f"Error parsing PDF: {e}"

def parse_txt(file_content):
    """Extracts text from TXT file content."""
    try:
        return file_content.decode('utf-8')
    except Exception as e:
        return f"Error parsing TXT: {e}"

# (CO1) Define skill keywords for extraction
SKILL_KEYWORDS = [
    'python', 'java', 'c++', 'javascript', 'sql', 'mysql', 'postgresql', 'mongodb',
    'react', 'angular', 'vue', 'node.js', 'django', 'flask', 'spring boot',
    'pandas', 'numpy', 'scipy', 'matplotlib', 'seaborn', 'scikit-learn', 'tensorflow',
    'keras', 'pytorch', 'machine learning', 'deep learning', 'data analysis', 'nlp',
    'aws', 'azure', 'google cloud', 'gcp', 'docker', 'kubernetes', 'git', 'jira',
    'agile', 'scrum', 'project management', 'product management', 'marketing', 'seo',
    'finance', 'accounting', 'excel', 'powerpoint', 'word', 'power bi', 'tableau',
```

```
    'figma', 'adobe xd', 'user research', 'prototyping', 'ui/ux', 'design'
]

EDUCATION_LEVELS = {
    'bachelor': ['bachelor', 'b.s', 'b.a', 'btech', 'b.e.'],
    'master': ['master', 'm.s', 'm.a', 'mtech', 'm.e.', 'mba'],
    'phd': ['phd', 'doctorate']
}

ROLE_KEYWORDS = {
    'Backend Developer': ['django', 'flask', 'spring boot', 'node.js', 'postgresql', 'mongodb'],
    'Frontend Developer': ['react', 'angular', 'vue', 'javascript', 'typescript'],
    'Data Scientist': ['python', 'machine learning', 'deep learning', 'tensorflow', 'pytorch', 'pandas',
'nlp'],
    'UI/UX Designer': ['figma', 'adobe xd', 'user research', 'prototyping', 'ui/ux'],
    'Manager': ['project management', 'product management', 'agile', 'scrum']
}

def estimate_role(skills):
    """Estimates role based on skill keywords."""
    role_scores = {role: 0 for role in ROLE_KEYWORDS}
    for skill in skills:
        for role, keywords in ROLE_KEYWORDS.items():
            if skill in keywords:
                role_scores[role] += 1

    if not any(role_scores.values()):
        return 'Generalist'

    # Return role with the highest score
    return max(role_scores, key=role_scores.get)

# --- NEW: Linear Regression Model for Salary ---

def train_salary_model():
    """
    Trains a simple Linear Regression model on dummy data.
    In a real system, you'd load a real salary dataset here.
    """
    print("Training dummy salary prediction model...")
    # 1. Create dummy data
    roles = ['Backend Developer', 'Frontend Developer', 'Data Scientist', 'UI/UX Designer',
'Manager', 'Generalist']
    data = []
```

```
for _ in range(50):
    role = random.choice(roles)
    exp = random.randint(0, 15)

    # Base salaries (from your old function)
    base_salaries = {
        'Backend Developer': 80000, 'Frontend Developer': 75000,
        'Data Scientist': 90000, 'UI/UX Designer': 65000,
        'Manager': 100000, 'Generalist': 50000
    }
    base = base_salaries[role]
    salary = base + exp * random.randint(2000, 3000) + random.randint(-5000, 5000)
    data.append([role, exp, salary])

df = pd.DataFrame(data, columns=['Role', 'Experience', 'Salary'])

# 2. Prepare data for ML
X_role = df[['Role']]
X_exp = df[['Experience']]
y = df['Salary']

# One-hot encode the 'Role'
encoder = OneHotEncoder(handle_unknown='ignore')
X_role_encoded = encoder.fit_transform(X_role)

# Combine encoded roles and experience
# Note: We create a simple model for demonstration.
# We'll actually just train on Experience for simplicity here.
# A real model would combine these features.

# Simple model: Regress salary on experience
X_train = df[['Experience']]
y_train = df['Salary']

# 3. Train the model
model = LinearRegression()
model.fit(X_train, y_train)

print("Salary model trained.")
# In a real app, you'd save this model, but for Colab, we just return it.
# For this demo, we'll return a dictionary of a model AND the base salaries
# to blend the ML approach with the rule-based approach.
return model, base_salaries, encoder # Returning base salaries and encoder for use
```

```
# --- THIS FUNCTION IS NOW FIXED ---
def estimate_salary_ml(model_components, role, experience):
    """Predicts salary using the trained Linear Regression model."""
    model, base_salaries, encoder = model_components

    # Use the base salary for the role as the starting point
    base = base_salaries.get(role, 50000)

    # --- FIX: Create a DataFrame with the same feature name as the training data ---
    exp_df = pd.DataFrame([[experience]], columns=['Experience'])

    # Get the model's coefficient (how much $ per year of exp)
    exp_coeff = model.coef_[0]

    # Calculate salary: Base for role + (coeff * experience) + randomness
    salary = base + (exp_coeff * experience) + random.randint(-5000, 5000)

    return int(salary)

# --- END NEW ---


def extract_info(text, salary_model): # <-- MODIFIED: Pass in the model
    """Extracts skills, experience, and education from cleaned text."""
    text_lower = text.lower()

    # 1. Extract Skills
    extracted_skills = set()
    for skill in SKILL_KEYWORDS:
        if re.search(r'\b' + re.escape(skill) + r'\b', text_lower):
            extracted_skills.add(skill)

    # 2. Extract Experience (simple regex, can be improved)
    years_exp = 0
    # Try to find "X+ years of experience"
    matches = re.findall(r'(\d+)\+?\s*years?(\s*of\s*experience)?', text_lower)
    if matches:
        years_exp = max([int(m[0]) for m in matches])
    else:
        # Try to find "X-Y years"
        matches = re.findall(r'(\d+)\s*-\s*(\d+)\s*years', text_lower)
        if matches:
            years_exp = max([int(m[1]) for m in matches])
```

```
    # 3. Extract Education
    extracted_edu = 'unknown'
    if any(keyword in text_lower for keyword in EDUCATION_LEVELS['phd']):
        extracted_edu = 'phd'
    elif any(keyword in text_lower for keyword in EDUCATION_LEVELS['master']):
        extracted_edu = 'master'
    elif any(keyword in text_lower for keyword in EDUCATION_LEVELS['bachelor']):
        extracted_edu = 'bachelor'

    # 4. Resume Quality Check
    quality_score = 0
    if 'education' in text_lower: quality_score += 1
    if 'experience' in text_lower or 'work history' in text_lower: quality_score += 1
    if 'skills' in text_lower: quality_score += 1
    if 'summary' in text_lower or 'objective' in text_lower: quality_score += 1

    # --- MODIFIED: Add Role and Salary Estimation ---
    skills_list = list(extracted_skills)
    role = estimate_role(skills_list)

    # Call the NEW ML salary estimator
    salary = estimate_salary_ml(salary_model, role, years_exp)

    return {
        'skills': skills_list,
        'experience_years': years_exp,
        'education': extracted_edu,
        'quality_score': quality_score, # out of 4
        'role': role,
        'salary': salary
    }

def get_jd_skills(jd_text):
    """Extracts keywords from a job description."""
    cleaned_jd = clean_resume(jd_text)
    jd_skills = set()
    for skill in SKILL_KEYWORDS:
        if re.search(r'\b' + re.escape(skill) + r'\b', cleaned_jd):
            jd_skills.add(skill)
    # If no specific skills found, use all non-stopwords
    if not jd_skills:
        jd_skills = set(word_tokenize(cleaned_jd))
    return jd_skills
```

```
# --- NEW: Train the salary model ONCE when cell is run ---
salary_model_components = train_salary_model()
# ---

print("Functions defined.")

# --- Cell 5: (CO1, CO5) Scoring Function (for single screening) ---

def score_resume(resume_info, jd_skills, min_exp, min_edu):
    """Scores a resume based on the defined criteria."""
    score = 0

    # 1. Skill Match Score (50%)
    skill_match_percent = 0
    if jd_skills:
        common_skills = set(resume_info['skills']).intersection(jd_skills)
        if len(jd_skills) > 0: # Avoid division by zero
            skill_match_percent = len(common_skills) / len(jd_skills)
    score += 0.50 * skill_match_percent

    # 2. Experience Score (25%)
    exp_score = 0
    if resume_info['experience_years'] >= min_exp:
        exp_score = 1
    score += 0.25 * exp_score

    # 3. Education Score (15%)
    edu_score = 0
    edu_map = {'unknown': 0, 'bachelor': 1, 'master': 2, 'phd': 3}
    if edu_map.get(resume_info['education'], 0) >= edu_map.get(min_edu, 0):
        edu_score = 1
    score += 0.15 * edu_score

    # 4. Resume Quality Score (10%)
    quality_score = resume_info['quality_score'] / 4.0
    score += 0.10 * quality_score

    return score * 100 # Return as a percentage

print("Scoring function defined.")

# --- Cell 6: (CO5) UI - Step 1: Upload Resumes to Talent Pool ---
print("\n--- 🧠 Smart Resume Screening System ---")
```

```
print("STEP 1 (Option A): Upload individual resumes below to screen them AND add them to the Talent Pool.")

# --- NEW: Global Talent Pool ---
CANDIDATE_DATABASE = []
# ---

# --- Define UI Elements ---
file_uploader = widgets.FileUpload(
    accept='.pdf, .txt',
    multiple=True,  # <-- CHANGE: Allow multiple files
    description='Upload Resumes'
)
jd_input = widgets.Textarea(
    placeholder='Paste the job description here...',
    layout=Layout(width='90%', height='200px')
)
exp_slider = widgets.IntSlider(
    value=0, min=0, max=20, step=1,
    description='Min Experience:'
)
edu_dropdown = widgets.Dropdown(
    options=[('Any', 'unknown'), ('Bachelor', 'bachelor'), ('Master', 'master'), ('PhD', 'phd')],
    value='unknown',
    description='Min Education:'
)
screen_button = widgets.Button(
    description='Screen & Add to Pool', # <-- CHANGE: Updated button text
    button_style='success'
)
screen_output = widgets.Output()

def on_screen_button_clicked(b):
    with screen_output:
        screen_output.clear_output()

        # --- FIX: More specific error checking ---
        files_uploaded = file_uploader.value
        jd_pasted = jd_input.value

        if not files_uploaded and not jd_pasted:
            print("ERROR: Please upload at least one resume AND paste a job description.")
            return
        elif not files_uploaded:
```

```
    print("ERROR: No files were detected in the uploader. Please try uploading again.")
    print("(Tip: After selecting files, wait a few seconds before clicking 'Screen'.)")
    return
elif not jd_pasted:
    print("ERROR: The job description text box is empty. Please paste the JD.")
    return
# --- END FIX ---

print(f"Processing {len(files_uploaded)} resume(s)...")

# 1. Get Job Description info ONCE
jd_skills = get_jd_skills(jd_pasted) # Use the new variable
min_exp = exp_slider.value
min_edu = edu_dropdown.value

results = []
new_candidates_added = 0

# 2. Get the list of all uploaded files
uploaded_files = list(files_uploaded.values()) # Use the new variable

if not uploaded_files:
     print("No files found in uploader. Please try again.")
     return

# 3. Loop through each file, process, and score
for uploaded_file_data in uploaded_files:
    try:
        file_name = uploaded_file_data['metadata']['name']
        file_content = uploaded_file_data['content']
    except Exception as e:
        print(f"\n--- Error accessing file data: {e} ---")
        print("Skipping this file. Please try uploading it again.")
        continue

    # Parse Text
    resume_text = ""
    if file_name.endswith('.pdf'):
        resume_text = parse_pdf(file_content)
    elif file_name.endswith('.txt'):
        resume_text = parse_txt(file_content)

    if "Error parsing" in resume_text:
        print(f"\n--- Error parsing {file_name}: {resume_text} ---")
```

```
            continue

        # Clean and Extract Info
        cleaned_text = clean_resume(resume_text)
        # --- MODIFIED: Pass salary model in ---
        resume_info = extract_info(cleaned_text, salary_model_components)

        # Score Resume
        final_score = score_resume(resume_info, jd_skills, min_exp, min_edu)

        # Store results
        candidate_record = {
            'file_name': file_name,
            'score': final_score,
            'info': resume_info,
            'cleaned_text': cleaned_text # --- NEW: Store cleaned text for K-Means ---
        }
        results.append(candidate_record)

        # --- NEW: Add to Global Database ---
        # Check if candidate is already in DB to avoid duplicates
        if not any(c['file_name'] == file_name for c in CANDIDATE_DATABASE):
            CANDIDATE_DATABASE.append(candidate_record)
            new_candidates_added += 1
        # ---

    if not results:
        print("\nNo resumes were successfully processed.")
        return

    # 4. Sort results by score (highest first)
    sorted_results = sorted(results, key=lambda x: x['score'], reverse=True)

    # 5. Display Summary Report
    print(f"\n--- Screening Summary: {len(sorted_results)} Resumes Ranked ---")
    for i, r in enumerate(sorted_results):
        # Salary formatting is applied here
        print(f"  {i+1}. {r['file_name']} \t Score: {r['score']:.2f}% \t (Est. Role: {r['info']['role']}, Est.
Salary: ${r['info']['salary']:,})")

    # 6. Display Detailed Reports
    print("\n\n--- Detailed Reports (Individual Analysis) ---")
    for r in sorted_results:
        resume_info = r['info'] # Get info from stored results
```

```
            print(f"\n----------------------------------------------")
            print(f"--- Screening Report for: {r['file_name']} ---")
            print(f"----------------------------------------------")

            print(f"\nOverall Match Score: {r['score']:.2f}%")

            print("\n--- Extracted Details (CO1) ---")
            print(f"Estimated Role: {resume_info['role']}")
            # Salary formatting is applied here
            print(f"Estimated Salary: ${resume_info['salary']:,} (Predicted by ML Model)")
            print(f"Experience: ~{resume_info['experience_years']} years (Min required: {min_exp})")
            print(f"Education: {resume_info['education'].capitalize()} (Min required:
{min_edu.capitalize()})")
            print(f"Resume Quality: {resume_info['quality_score']}/4")

            common_skills = set(resume_info['skills']).intersection(jd_skills)
            print(f"\nSkill Match: {len(common_skills)} / {len(jd_skills)} keywords")
            print(f"  - Matched Skills: {list(common_skills) if common_skills else 'None'}")
            print(f"  - All Extracted Skills: {resume_info['skills'] if resume_info['skills'] else 'None'}")

        print("\n" + "="*50)
        print(f"[!] {new_candidates_added} new candidates added to the Talent Pool.")
        print(f"[!] Total candidates in Talent Pool: {len(CANDIDATE_DATABASE)}")
        print("="*50)

        # Reset the uploader
        file_uploader.value.clear()
        file_uploader._counter += 1


screen_button.on_click(on_screen_button_clicked)

# --- Assemble and Display the UI ---
ui_step1 = VBox([
    widgets.HTML("<h2>(CO5) UI Step 1: Upload & Screen Resumes</h2>"),
    widgets.HTML("<h4>This populates the 'Talent Pool' for the Team Builder.</h4>"),
    file_uploader,
    widgets.HTML("<h4>Job Description (for screening):</h4>"),
    jd_input,
    widgets.HTML("<h4>Minimum Requirements (for screening):</h4>"),
    HBox([exp_slider, edu_dropdown]),
    screen_button,
    screen_output
```

```
])

display(ui_step1)

# --- MODIFIED Cell 7: Screen & Load Talent Pool from File UI ---
print("\n\n--- STEP 1.2 (Option B): Screen & Load Talent Pool from File ---")
print("Or, upload a .csv or .xlsx file to screen many candidates at once.")

# --- Define UI Elements ---
dataset_uploader = widgets.FileUpload(
    accept='.csv, .xlsx',
    multiple=False,
    description='Upload Dataset'
)
text_column_input = widgets.Text(
    value='Resume', # Pre-fill with the column name from your example
    placeholder='e.g., Resume, text, resume_text',
    description='Resume Text Column:',
    layout=Layout(width='50%')
)

# --- NEW WIDGETS ---
jd_input_batch = widgets.Textarea(
    placeholder='Paste the job description here...',
    layout=Layout(width='90%', height='200px')
)
exp_slider_batch = widgets.IntSlider(
    value=0, min=0, max=20, step=1,
    description='Min Experience:'
)
edu_dropdown_batch = widgets.Dropdown(
    options=[('Any', 'unknown'), ('Bachelor', 'bachelor'), ('Master', 'master'), ('PhD', 'phd')],
    value='unknown',
    description='Min Education:'
)
# --- END NEW ---

load_dataset_button = widgets.Button(
    description='Screen & Load to Pool',
    button_style='primary'
)
dataset_output = widgets.Output()

def on_load_dataset_button_clicked(b):
```

```
with dataset_output:
    dataset_output.clear_output()

    uploaded_file_data = dataset_uploader.value
    text_col = text_column_input.value

    if not uploaded_file_data:
        print("ERROR: Please upload a .csv or .xlsx file.")
        return
    if not text_col:
        print("ERROR: Please specify the name of the column containing the resume text.")
        return

    # Get the first (and only) file
    try:
        file_data = list(uploaded_file_data.values())[0]
        file_name = file_data['metadata']['name']
        file_content = file_data['content']
    except Exception as e:
        print(f"Error accessing file: {e}")
        return

    print(f"Loading and processing file: {file_name}...")

    # Read file into pandas
    df = None
    try:
        if file_name.endswith('.csv'):
            df = pd.read_csv(io.BytesIO(file_content))
        elif file_name.endswith('.xlsx'):
            df = pd.read_excel(io.BytesIO(file_content))
    except Exception as e:
        print(f"Error reading file: {e}. Make sure it's a valid CSV or Excel file.")
        return

    # Check if text column exists
    if text_col not in df.columns:
        print(f"ERROR: Column '{text_col}' not found in the file.")
        print(f"Available columns are: {list(df.columns)}")
        return

    # --- NEW: Get JD and Requirements ---
    jd_pasted = jd_input_batch.value
    if not jd_pasted:
```

```
    print("ERROR: The job description text box is empty. Please paste the JD.")
    return

jd_skills = get_jd_skills(jd_pasted)
min_exp = exp_slider_batch.value
min_edu = edu_dropdown_batch.value
# --- END NEW ---

print(f"Found {len(df)} candidates. Processing, screening, and adding to Talent Pool...")

new_candidates_added = 0
results_batch = [] # <-- NEW: To store results for ranking

for index, row in df.iterrows():
    resume_text = str(row[text_col]) # Ensure text is string

    # Use index or another column as a unique ID
    # --- MODIFIED: Use a more descriptive ID ---
    candidate_id = f"file_{file_name}row{index}"

    # Check for duplicates before processing
    if not any(c['file_name'] == candidate_id for c in CANDIDATE_DATABASE):
        # Clean and Extract Info
        cleaned_text = clean_resume(resume_text)
        # --- MODIFIED: Pass salary model in ---
        resume_info = extract_info(cleaned_text, salary_model_components)

        # --- NEW: Perform Screening ---
        final_score = score_resume(resume_info, jd_skills, min_exp, min_edu)
        # --- END NEW ---

        # Store results
        candidate_record = {
            'file_name': candidate_id, # Use the descriptive ID
            'score': final_score, # <-- MODIFIED: Was 0
            'info': resume_info,
            'cleaned_text': cleaned_text # --- NEW: Store cleaned text for K-Means ---
        }

        CANDIDATE_DATABASE.append(candidate_record)
        results_batch.append(candidate_record) # <-- NEW
        new_candidates_added += 1

# --- NEW: Print Summary Report ---
```

```
        if not results_batch:
            print("\nNo new candidates were processed from the file (they may already be in the
Talent Pool).")
        else:
            sorted_results_batch = sorted(results_batch, key=lambda x: x['score'], reverse=True)
            print(f"\n--- Screening Summary: {len(sorted_results_batch)} Resumes Ranked ---")
            for i, r in enumerate(sorted_results_batch):
                # Salary formatting is applied here
                print(f"  {i+1}. {r['file_name']} \t Score: {r['score']:.2f}% \t (Est. Role: {r['info']['role']}, Est.
Salary: ${r['info']['salary']:,})")
        # --- END NEW ---

        print("\n" + "="*50)
        print(f"[!] {new_candidates_added} new candidates loaded from file.")
        print(f"[!] Total candidates in Talent Pool: {len(CANDIDATE_DATABASE)}")
        print("="*50)

        # Reset the uploader
        dataset_uploader.value.clear()
        dataset_uploader._counter += 1


load_dataset_button.on_click(on_load_dataset_button_clicked)

# --- Assemble and Display the UI ---
ui_step1_2 = VBox([
    widgets.HTML("<h2>(NEW) UI Step 1.2: Screen & Load Talent Pool from File</h2>"),
    widgets.HTML("<h4>Upload a CSV or Excel file of resumes to screen them all at
once.</h4>"),
    HBox([dataset_uploader, text_column_input]),
    # --- NEW ---
    widgets.HTML("<h4>Job Description (for screening):</h4>"),
    jd_input_batch,
    widgets.HTML("<h4>Minimum Requirements (for screening):</h4>"),
    HBox([exp_slider_batch, edu_dropdown_batch]),
    # --- END NEW ---
    load_dataset_button,
    dataset_output
])

display(ui_step1_2)

# --- MODIFIED Cell 8: Talent Pool Analyzer UI (with K-Means Clustering) ---
print("\n\n--- STEP 1.5: Talent Pool Analysis ---")
```

```
print("Analyze the skills of all candidates currently in the Talent Pool.")

# --- Define UI Elements ---
analyze_pool_button = widgets.Button(
    description='Analyze Talent Pool',
    button_style='info'
)
analysis_output = widgets.Output()

def on_analyze_pool_button_clicked(b):
    with analysis_output:
        analysis_output.clear_output()

        if not CANDIDATE_DATABASE:
            print("ERROR: The Talent Pool is empty. Please upload resumes in Step 1 or 1.2 first.")
            return

        # --- MODIFIED: More robust check for K-Means ---
        if len(CANDIDATE_DATABASE) < 2:
             print("ERROR: Need at least 2 candidates in the pool to run analysis.")
             return

        print(f"--- Analyzing {len(CANDIDATE_DATABASE)} Candidates in Talent Pool ---")

        all_skills_list = []
        all_skill_sets = []

        for candidate in CANDIDATE_DATABASE:
            skills = candidate['info']['skills']
            all_skills_list.extend(skills)
            all_skill_sets.append(set(skills))

        # 1. Find Top 10 Most Common Skills
        skill_counts = Counter(all_skills_list)
        print("\n--- Top 10 Most Common Skills ---")
        if not skill_counts:
            print("No skills found in the pool.")
        else:
            for skill, count in skill_counts.most_common(10):
                print(f"  - {skill} (Appears in {count} resumes)")

        # 2. Find "Must-Have" Skills (Present in ALL resumes)
        print("\n--- Skills Present in ALL Resumes ---")
        if not all_skill_sets:
```

```
    print("No skills found.")
else:
    common_to_all = set.intersection(*all_skill_sets)
    if not common_to_all:
        print("No single skill was found in ALL resumes.")
    else:
        print(f"  {list(common_to_all)}")

# --- NEW: K-Means Clustering Analysis ---
print("\n--- Candidate Clusters (K-Means Clustering) ---")

# 1. Get the text corpus from the database
corpus = [c['cleaned_text'] for c in CANDIDATE_DATABASE]

# 2. Vectorize the text
# We limit features to 100 to get broad topics
vectorizer = TfidfVectorizer(max_features=100, stop_words='english')
X = vectorizer.fit_transform(corpus)

# 3. Run K-Means
# Set n_clusters, but not more than the number of candidates
num_candidates = len(CANDIDATE_DATABASE)
n_clusters = min(4, num_candidates) # Use 4 clusters, or fewer if not enough candidates

if n_clusters <= 1:
    print("Not enough candidates to form clusters.")
    return

kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)
kmeans.fit(X)

clusters = kmeans.labels_

# 4. Group candidates by their assigned cluster
clustered_candidates = {i: [] for i in range(n_clusters)}
for i, candidate in enumerate(CANDIDATE_DATABASE):
    cluster_num = clusters[i]
    clustered_candidates[cluster_num].append(candidate['file_name'])

print("\n--- Cluster Definitions (Top Keywords) ---")
# Show top keywords for each cluster
order_centroids = kmeans.cluster_centers_.argsort()[:, ::-1]
terms = vectorizer.get_feature_names_out()
```

```
        for i in range(n_clusters):
            top_words = [terms[ind] for ind in order_centroids[i, :5]]
            print(f"Cluster {i}: {', '.join(top_words)}")

        print("\n--- Candidates by Cluster ---")
        for cluster_num, candidates in clustered_candidates.items():
            print(f"Cluster {cluster_num} ({len(candidates)} candidates):")
            for name in candidates:
                print(f"  - {name}")
        # --- END NEW ---

        print("\n" + "="*50)


analyze_pool_button.on_click(on_analyze_pool_button_clicked)

# --- Assemble and Display the UI ---
ui_step1_5 = VBox([
    widgets.HTML("<h2>UI Step 1.5: Analyze Talent Pool</h2>"),
    widgets.HTML("<h4>Click to find common skills and run K-Means clustering.</h4>"),
    analyze_pool_button,
    analysis_output
])

display(ui_step1_5)

# --- Cell 9: (CO1) Team Assembly (Heuristic Search) ---
print("\n\n--- STEP 2: Team Assembly (Greedy Heuristic Search) ---")
print("Define your project needs, and the algorithm will build the best team.")

# --- NEW: This is our Heuristic Function ---
def calculate_candidate_fitness(candidate_info, required_role):
    """
    This is our heuristic function h(n) to score a candidate.
    Higher score is better.
    """
    fitness_score = 0

    # 1. Role Match Bonus (High Importance)
    if candidate_info['role'].lower() == required_role.lower():
        fitness_score += 100

    # 2. Experience Bonus
    fitness_score += candidate_info['experience_years'] * 5 # 5 points per year
```

```
    # 3. Skill Bonus
    fitness_score += len(candidate_info['skills']) # 1 point per skill

    # 4. Cost (Salary) "Bonus" - lower is better
    # We invert the salary, so a lower salary gives a higher score.
    # We add 1 to avoid division by zero and scale it.
    salary_score = 10000 / (candidate_info['salary'] + 1)
    fitness_score += salary_score

    return fitness_score
# --- END NEW ---


# --- MODIFIED: This is the Greedy Best-First Search Algorithm ---
def on_find_team_button_clicked(b):
    with team_output:
        team_output.clear_output()

        if not CANDIDATE_DATABASE:
            print("ERROR: The Talent Pool is empty. Please upload resumes first.")
            return

        # 1. Get Project Requirements
        try:
            # We just need the roles for this search
            required_roles = [r.strip().lower() for r in team_roles_input.value.split(',') if r.strip()]
        except Exception as e:
            print(f"Error parsing inputs: {e}. Please check formatting.")
            return

        if not required_roles:
            print("ERROR: Please enter at least one role to build a team (e.g., Data Scientist,
Manager).")
            return

        print(f"--- Running Greedy Heuristic Search for {len(required_roles)} Roles ---")

        # 2. Create a copy of the database to use as the "available" pool
        available_candidates = list(CANDIDATE_DATABASE)
        final_team = []
        team_salary = 0
        total_fitness = 0
```

```
# 3. The Greedy Search Loop
for role in required_roles:
    best_candidate = None
    best_score = -1 # Start at -1

    # 4. Heuristic Search: Find the best candidate for this role
    for candidate in available_candidates:
        # Calculate the heuristic score for this candidate/role combo
        score = calculate_candidate_fitness(candidate['info'], role)

        if score > best_score:
            best_score = score
            best_candidate = candidate

    # 5. Greedy Selection
    if best_candidate:
        final_team.append((role, best_candidate, best_score))
        team_salary += best_candidate['info']['salary']
        total_fitness += best_score
        # Remove from pool so they can't be picked again
        available_candidates.remove(best_candidate)
    else:
        # No one was found for this role
        final_team.append((role, None, 0))

# 6. Print Recommendation
print("\n" + "="*50)
print("--- Optimal Team Recommendation ---")
print("="*50)

for role, candidate, score in final_team:
    if candidate:
        info = candidate['info']
        print(f"\n  ✅ {role.upper()}: {candidate['file_name']}")
        print(f"     (Role: {info['role']}, Exp: {info['experience_years']} yrs)")
        print(f"     (Salary: ${info['salary']:,}, Fitness Score: {score:.2f})")
    else:
        print(f"\n  ❌ {role.upper()}: No candidate found in pool.")

print("\n" + "="*50)
print("--- Team Summary ---")
print(f"Total Fitness Score: {total_fitness:.2f}")
print(f"Total Estimated Salary: ${team_salary:,}")
print("="*50)
```

```
# --- End of Search Logic ---


# --- MODIFIED: Simplified UI for the new search ---
team_roles_input = widgets.Text(
    placeholder='e.g., Backend Developer, Data Scientist, Manager',
    description='Team Roles (csv):',
    layout=Layout(width='90%')
)
find_team_button = widgets.Button(
    description='Build Optimal Team', # <-- New button text
    button_style='success'
)
team_output = widgets.Output()

# --- Attach the logic to the button ---
find_team_button.on_click(on_find_team_button_clicked)
print("\nTeam Builder (Heuristic Search) logic is defined and attached.")


# --- Finally, assemble and display the UI ---
ui_step2 = VBox([
    widgets.HTML("<h2>(CO1) UI Step 2: Build Optimal Team</h2>"),
    widgets.HTML("<h4>Define the roles you need. The algorithm will find the best candidate for
each.</h4>"),
    team_roles_input,
    find_team_button,
    team_output
])

display(ui_step2)
```