"""

TalentIQ - Advanced Recruitment Intelligence Platform

A comprehensive AI-powered recruitment system with advanced analytics,

talent mapping, and predictive hiring insights.

"""

import streamlit as st

st.set\_page\_config(

page\_title="TalentIQ™ | Smart Recruitment Platform",

layout="wide",

initial\_sidebar\_state="expanded",

menu\_items={

'About': "TalentIQ - Next-Gen Recruitment Intelligence Platform"

}

)

import io, os, re, sqlite3, json, hashlib

from datetime import datetime, timedelta

from typing import List, Dict, Tuple, Optional, Any

from dataclasses import dataclass, field

from enum import Enum

import math

from concurrent.futures import ThreadPoolExecutor

import logging

# Configure logging

logging.basicConfig(level=logging.INFO)

logger = logging.getLogger(\_\_name\_\_)

# ----- Enhanced Import Management -----

class DependencyManager:

"""Manages optional dependencies with graceful fallbacks"""

def \_\_init\_\_(self):

self.dependencies = {}

self.missing = []

self.\_load\_dependencies()

def \_load\_dependencies(self):

deps = {

'pdfplumber': None,

'docx2txt': None,

'rapidfuzz': None,

'sentence\_transformers': None,

'pandas': None,

'plotly': None,

'fpdf': None,

'numpy': None,

'sklearn': None

}

for dep\_name in deps:

try:

if dep\_name == 'sentence\_transformers':

from sentence\_transformers import SentenceTransformer, util

self.dependencies['SentenceTransformer'] = SentenceTransformer

self.dependencies['util'] = util

elif dep\_name == 'rapidfuzz':

from rapidfuzz import fuzz

self.dependencies['fuzz'] = fuzz

elif dep\_name == 'plotly':

import plotly.graph\_objects as go

import plotly.express as px

self.dependencies['go'] = go

self.dependencies['px'] = px

elif dep\_name == 'sklearn':

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

self.dependencies['TfidfVectorizer'] = TfidfVectorizer

self.dependencies['cosine\_similarity'] = cosine\_similarity

else:

module = \_\_import\_\_(dep\_name)

self.dependencies[dep\_name] = module

except ImportError:

self.missing.append(dep\_name)

logger.warning(f"Optional dependency {dep\_name} not available")

def get(self, name):

return self.dependencies.get(name)

def check\_missing(self):

if self.missing:

return True, self.missing

return False, []

deps = DependencyManager()

# Check dependencies

has\_missing, missing\_deps = deps.check\_missing()

if has\_missing:

st.sidebar.warning(f"⚠️ Optional features unavailable: {', '.join(missing\_deps)}")

st.sidebar.caption(f"Install: `pip install {' '.join(missing\_deps)}`")

pd = deps.get('pandas')

pdfplumber = deps.get('pdfplumber')

docx2txt = deps.get('docx2txt')

fuzz = deps.get('fuzz')

SentenceTransformer = deps.get('SentenceTransformer')

util = deps.get('util')

go = deps.get('go')

px = deps.get('px')

FPDF = deps.get('fpdf')

np = deps.get('numpy')

TfidfVectorizer = deps.get('TfidfVectorizer')

cosine\_similarity = deps.get('cosine\_similarity')

# ----- Data Models -----

class MatchLevel(Enum):

EXCELLENT = "Excellent Match"

STRONG = "Strong Match"

GOOD = "Good Match"

MODERATE = "Moderate Match"

WEAK = "Weak Match"

@dataclass

class JobDescription:

id: int

title: str

department: str

raw\_text: str

must\_have\_skills: List[str]

good\_to\_have\_skills: List[str]

experience\_range: Tuple[int, int]

created\_at: datetime

metadata: Dict[str, Any] = field(default\_factory=dict)

@dataclass

class CandidateProfile:

name: str

email: str

phone: str

resume\_text: str

skills: List[str]

experience\_years: int

education: List[str]

certifications: List[str]

@dataclass

class EvaluationResult:

candidate: CandidateProfile

job: JobDescription

technical\_score: float

cultural\_fit\_score: float

experience\_score: float

semantic\_score: float

final\_score: float

match\_level: MatchLevel

strengths: List[str]

gaps: List[str]

recommendations: List[str]

interview\_questions: List[str]

risk\_factors: List[str]

# ----- Enhanced Database Layer -----

class DatabaseManager:

"""Enhanced database management with migration support"""

def \_\_init\_\_(self, db\_path: str = "talentiq.db"):

self.db\_path = db\_path

self.conn = None

self.init\_database()

def init\_database(self):

"""Initialize database with enhanced schema"""

self.conn = sqlite3.connect(self.db\_path, check\_same\_thread=False)

cur = self.conn.cursor()

# Enhanced JD table

cur.execute("""

CREATE TABLE IF NOT EXISTS job\_descriptions (

id INTEGER PRIMARY KEY AUTOINCREMENT,

title TEXT NOT NULL,

department TEXT,

company TEXT,

location TEXT,

employment\_type TEXT,

salary\_range TEXT,

raw\_text TEXT,

must\_have\_skills TEXT,

good\_to\_have\_skills TEXT,

experience\_min INTEGER,

experience\_max INTEGER,

metadata TEXT,

status TEXT DEFAULT 'active',

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

)""")

# Enhanced candidate evaluations

cur.execute("""

CREATE TABLE IF NOT EXISTS candidate\_evaluations (

id INTEGER PRIMARY KEY AUTOINCREMENT,

job\_id INTEGER,

candidate\_name TEXT,

candidate\_email TEXT,

candidate\_phone TEXT,

resume\_hash TEXT,

resume\_text TEXT,

technical\_score REAL,

cultural\_fit\_score REAL,

experience\_score REAL,

semantic\_score REAL,

final\_score REAL,

match\_level TEXT,

strengths TEXT,

gaps TEXT,

recommendations TEXT,

interview\_questions TEXT,

risk\_factors TEXT,

recruiter\_notes TEXT,

status TEXT DEFAULT 'pending',

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY (job\_id) REFERENCES job\_descriptions(id)

)""")

# Analytics table

cur.execute("""

CREATE TABLE IF NOT EXISTS recruitment\_analytics (

id INTEGER PRIMARY KEY AUTOINCREMENT,

job\_id INTEGER,

metric\_type TEXT,

metric\_value REAL,

metadata TEXT,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY (job\_id) REFERENCES job\_descriptions(id)

)""")

# Interview feedback

cur.execute("""

CREATE TABLE IF NOT EXISTS interview\_feedback (

id INTEGER PRIMARY KEY AUTOINCREMENT,

evaluation\_id INTEGER,

interviewer TEXT,

round TEXT,

rating INTEGER,

comments TEXT,

recommendation TEXT,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY (evaluation\_id) REFERENCES candidate\_evaluations(id)

)""")

self.conn.commit()

def \_\_del\_\_(self):

if self.conn:

self.conn.close()

db = DatabaseManager()

# ----- Advanced Text Processing -----

class TextProcessor:

"""Advanced text processing and extraction"""

SKILL\_PATTERNS = {

'programming': r'\b(python|java|c\+\+|c#|javascript|typescript|go|rust|ruby|php|swift|kotlin|scala|r)\b',

'frameworks': r'\b(react|angular|vue|django|flask|spring|express|fastapi|rails|laravel)\b',

'databases': r'\b(sql|mysql|postgresql|mongodb|redis|cassandra|elasticsearch|dynamodb)\b',

'cloud': r'\b(aws|azure|gcp|cloud|lambda|ec2|s3|kubernetes|docker|terraform)\b',

'ai\_ml': r'\b(machine learning|deep learning|ai|tensorflow|pytorch|scikit-learn|nlp|computer vision)\b',

'devops': r'\b(ci/cd|jenkins|gitlab|github actions|ansible|puppet|chef|monitoring|logging)\b'

}

EDUCATION\_PATTERNS = {

'degree': r'\b(bachelor|master|phd|mba|b\.?tech|m\.?tech|b\.?e|m\.?e|bsc|msc)\b',

'university': r'\b(university|college|institute|school)\b'

}

@staticmethod

def extract\_text\_from\_file(uploaded\_file) -> str:

"""Enhanced file extraction with better error handling"""

if not uploaded\_file:

return ""

filename = uploaded\_file.name.lower()

try:

data = uploaded\_file.read()

except:

try:

data = uploaded\_file.getvalue()

except:

logger.error(f"Failed to read file: {filename}")

return ""

extractors = {

'.pdf': TextProcessor.\_extract\_pdf,

'.docx': TextProcessor.\_extract\_docx,

'.txt': TextProcessor.\_extract\_text

}

for ext, extractor in extractors.items():

if filename.endswith(ext):

return extractor(data)

# Default to text extraction

return TextProcessor.\_extract\_text(data)

@staticmethod

def \_extract\_pdf(data: bytes) -> str:

if not pdfplumber:

return ""

try:

with pdfplumber.open(io.BytesIO(data)) as pdf:

text = "\n".join([p.extract\_text() or "" for p in pdf.pages])

return text

except Exception as e:

logger.error(f"PDF extraction error: {e}")

return ""

@staticmethod

def \_extract\_docx(data: bytes) -> str:

if not docx2txt:

return ""

temp\_file = f"temp\_{datetime.now().timestamp()}.docx"

try:

with open(temp\_file, "wb") as f:

f.write(data)

text = docx2txt.process(temp\_file)

return text or ""

except Exception as e:

logger.error(f"DOCX extraction error: {e}")

return ""

finally:

if os.path.exists(temp\_file):

os.remove(temp\_file)

@staticmethod

def \_extract\_text(data: bytes) -> str:

try:

return data.decode('utf-8', errors='ignore')

except:

return ""

@staticmethod

def clean\_and\_normalize(text: str) -> str:

"""Advanced text cleaning and normalization"""

if not text:

return ""

# Remove excessive whitespace

text = re.sub(r'\s+', ' ', text)

# Remove special characters but keep important ones

text = re.sub(r'[^\w\s\-\.\,\;\:\@\#\+]', ' ', text)

# Normalize case for better matching

lines = text.split('\n')

normalized = []

for line in lines:

line = line.strip()

if not line or len(line) < 3:

continue

# Skip repetitive headers/footers

if line.lower() in ['page', 'resume', 'cv', 'curriculum vitae']:

continue

normalized.append(line)

return ' '.join(normalized)

@staticmethod

def extract\_contact\_info(text: str) -> Dict[str, str]:

"""Extract contact information from resume"""

contact = {

'email': '',

'phone': '',

'linkedin': '',

'github': ''

}

# Email pattern

email\_match = re.search(r'\b[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', text)

if email\_match:

contact['email'] = email\_match.group()

# Phone pattern (various formats)

phone\_match = re.search(r'(\+\d{1,3}[-.\s]?)?\(?\d{3,4}\)?[-.\s]?\d{3,4}[-.\s]?\d{3,4}', text)

if phone\_match:

contact['phone'] = phone\_match.group()

# LinkedIn

linkedin\_match = re.search(r'linkedin\.com/in/[\w\-]+', text, re.IGNORECASE)

if linkedin\_match:

contact['linkedin'] = linkedin\_match.group()

# GitHub

github\_match = re.search(r'github\.com/[\w\-]+', text, re.IGNORECASE)

if github\_match:

contact['github'] = github\_match.group()

return contact

@staticmethod

def extract\_experience\_years(text: str) -> int:

"""Extract years of experience from resume"""

patterns = [

r'(\d+)\+?\s\*years?\s\*(?:of\s\*)?experience',

r'experience[:\s]+(\d+)\+?\s\*years?',

r'(\d+)\s\*years?\s\*(?:of\s\*)?professional'

]

for pattern in patterns:

match = re.search(pattern, text, re.IGNORECASE)

if match:

return int(match.group(1))

# Fallback: count years mentioned in employment history

year\_mentions = re.findall(r'\b(19|20)\d{2}\b', text)

if len(year\_mentions) >= 2:

years = [int(y) for y in year\_mentions]

return max(years) - min(years)

return 0

@staticmethod

def extract\_skills(text: str) -> Dict[str, List[str]]:

"""Extract categorized skills from text"""

skills = {}

text\_lower = text.lower()

for category, pattern in TextProcessor.SKILL\_PATTERNS.items():

matches = re.findall(pattern, text\_lower)

if matches:

skills[category] = list(set(matches))

return skills

@staticmethod

def extract\_education(text: str) -> List[str]:

"""Extract education information"""

education = []

text\_lower = text.lower()

for match in re.finditer(TextProcessor.EDUCATION\_PATTERNS['degree'], text\_lower):

start = max(0, match.start() - 100)

end = min(len(text), match.end() + 100)

context = text[start:end]

education.append(context.strip())

return education

# ----- Advanced Matching Engine -----

class MatchingEngine:

"""Advanced candidate-job matching with multiple algorithms"""

def \_\_init\_\_(self):

self.embedder = self.\_load\_embedder()

self.tfidf = TfidfVectorizer(max\_features=500, ngram\_range=(1, 3)) if TfidfVectorizer else None

def \_load\_embedder(self):

"""Load sentence transformer model with caching"""

if not SentenceTransformer:

return None

try:

return SentenceTransformer('all-MiniLM-L6-v2')

except Exception as e:

logger.error(f"Failed to load embedding model: {e}")

return None

def calculate\_technical\_match(self, jd\_skills: List[str], candidate\_skills: List[str]) -> Tuple[float, List[str], List[str]]:

"""Calculate technical skills match with fuzzy matching"""

if not jd\_skills:

return 0.0, [], []

matched = []

unmatched = []

for jd\_skill in jd\_skills:

found = False

jd\_lower = jd\_skill.lower()

# Exact match

for c\_skill in candidate\_skills:

if jd\_lower == c\_skill.lower():

matched.append(jd\_skill)

found = True

break

# Fuzzy match

if not found and fuzz:

for c\_skill in candidate\_skills:

if fuzz.ratio(jd\_lower, c\_skill.lower()) >= 85:

matched.append(jd\_skill)

found = True

break

if not found:

unmatched.append(jd\_skill)

score = (len(matched) / len(jd\_skills)) \* 100 if jd\_skills else 0

return round(score, 2), matched, unmatched

def calculate\_semantic\_similarity(self, jd\_text: str, resume\_text: str) -> float:

"""Calculate semantic similarity using embeddings or TF-IDF"""

if not jd\_text or not resume\_text:

return 0.0

# Try embeddings first

if self.embedder and util:

try:

jd\_emb = self.embedder.encode(jd\_text[:2000], convert\_to\_tensor=True)

resume\_emb = self.embedder.encode(resume\_text[:2000], convert\_to\_tensor=True)

similarity = util.pytorch\_cos\_sim(jd\_emb, resume\_emb).item() \* 100

return round(similarity, 2)

except:

pass

# Fallback to TF-IDF

if self.tfidf and cosine\_similarity:

try:

vectors = self.tfidf.fit\_transform([jd\_text[:2000], resume\_text[:2000]])

similarity = cosine\_similarity(vectors[0:1], vectors[1:2])[0][0] \* 100

return round(similarity, 2)

except:

pass

# Basic keyword overlap

jd\_words = set(jd\_text.lower().split())

resume\_words = set(resume\_text.lower().split())

overlap = len(jd\_words & resume\_words) / len(jd\_words) if jd\_words else 0

return round(overlap \* 100, 2)

def calculate\_experience\_match(self, required\_range: Tuple[int, int], candidate\_years: int) -> float:

"""Calculate experience match score"""

min\_years, max\_years = required\_range

if candidate\_years < min\_years:

# Under-qualified

score = max(0, 100 - (min\_years - candidate\_years) \* 20)

elif candidate\_years > max\_years:

# Over-qualified (slight penalty)

score = max(70, 100 - (candidate\_years - max\_years) \* 5)

else:

# Perfect match

score = 100

return round(score, 2)

def calculate\_cultural\_fit(self, resume\_text: str) -> float:

"""Estimate cultural fit based on soft skills and values"""

cultural\_keywords = {

'teamwork': ['team', 'collaboration', 'cooperat', 'together'],

'leadership': ['lead', 'manage', 'mentor', 'guide', 'coach'],

'innovation': ['innovat', 'creative', 'novel', 'pioneer', 'transform'],

'communication': ['communicat', 'present', 'document', 'articulate'],

'problem\_solving': ['problem', 'solution', 'analytical', 'troubleshoot'],

'adaptability': ['adapt', 'flexible', 'agile', 'versatile'],

'growth': ['learn', 'grow', 'develop', 'improve', 'progress']

}

text\_lower = resume\_text.lower()

score = 0

max\_score = len(cultural\_keywords) \* 10

for category, keywords in cultural\_keywords.items():

for keyword in keywords:

if keyword in text\_lower:

score += 10

break

return round((score / max\_score) \* 100, 2)

def determine\_match\_level(self, score: float) -> MatchLevel:

"""Determine match level based on final score"""

if score >= 85:

return MatchLevel.EXCELLENT

elif score >= 75:

return MatchLevel.STRONG

elif score >= 65:

return MatchLevel.GOOD

elif score >= 50:

return MatchLevel.MODERATE

else:

return MatchLevel.WEAK

def generate\_risk\_factors(self, candidate: CandidateProfile, job: JobDescription, gaps: List[str]) -> List[str]:

"""Identify potential risk factors"""

risks = []

# Critical skill gaps

critical\_gaps = [g for g in gaps if g in job.must\_have\_skills]

if len(critical\_gaps) > 3:

risks.append(f"Missing {len(critical\_gaps)} critical skills")

# Experience mismatch

if candidate.experience\_years < job.experience\_range[0]:

deficit = job.experience\_range[0] - candidate.experience\_years

risks.append(f"Experience deficit of {deficit} years")

elif candidate.experience\_years > job.experience\_range[1] + 5:

risks.append("Significantly overqualified - retention risk")

# Job hopping

if 'month' in candidate.resume\_text.lower():

month\_counts = len(re.findall(r'\d+\s\*months?', candidate.resume\_text.lower()))

if month\_counts > 3:

risks.append("Potential job hopping pattern detected")

return risks

# ----- Interview Intelligence -----

class InterviewIntelligence:

"""Generate intelligent interview questions and assessment criteria"""

QUESTION\_TEMPLATES = {

'technical': {

'python': [

"Explain the difference between deep and shallow copy in Python. Provide examples.",

"How would you optimize a Python application that's running slowly?",

"Describe decorators and their use cases with a practical example."

],

'javascript': [

"Explain event delegation and bubbling in JavaScript.",

"What are closures and how do they work? Provide a use case.",

"How would you handle asynchronous operations in modern JavaScript?"

],

'system\_design': [

"Design a scalable URL shortening service like bit.ly",

"How would you architect a real-time chat application?",

"Explain how you'd design a distributed caching system."

],

'database': [

"Explain database indexing strategies and their trade-offs.",

"How would you optimize a slow-running query?",

"Describe ACID properties with real-world examples."

]

},

'behavioral': [

"Describe a challenging technical problem you solved. What was your approach?",

"Tell me about a time you had to learn a new technology quickly.",

"How do you handle disagreements with team members about technical decisions?",

"Describe a situation where you had to balance technical debt with feature delivery."

],

'situational': [

"Your team's deployment fails in production. How do you handle it?",

"A stakeholder requests a feature that you know is technically unfeasible. How do you respond?",

"You discover a security vulnerability in legacy code. What steps do you take?"

]

}

@classmethod

def generate\_questions(cls, skills\_gap: List[str], matched\_skills: List[str],

seniority\_level: str = "mid") -> List[str]:

"""Generate personalized interview questions"""

questions = []

# Technical questions based on matched skills

for skill in matched\_skills[:3]:

skill\_lower = skill.lower()

for category, q\_list in cls.QUESTION\_TEMPLATES['technical'].items():

if category in skill\_lower or skill\_lower in category:

questions.extend(q\_list[:1])

break

# Gap assessment questions

for gap in skills\_gap[:2]:

questions.append(f"How would you approach learning {gap}? What resources would you use?")

# Add behavioral questions

questions.extend(cls.QUESTION\_TEMPLATES['behavioral'][:2])

# Add situational questions based on seniority

if seniority\_level in ["senior", "lead"]:

questions.extend(cls.QUESTION\_TEMPLATES['situational'][:2])

return questions[:8] # Limit to 8 questions

# ----- Analytics Engine -----

class AnalyticsEngine:

"""Advanced analytics and insights generation"""

@staticmethod

def generate\_batch\_insights(evaluations: pd.DataFrame) -> Dict[str, Any]:

"""Generate insights from batch evaluations"""

if evaluations.empty:

return {}

insights = {

'total\_candidates': len(evaluations),

'avg\_match\_score': evaluations['final\_score'].mean(),

'match\_distribution': evaluations['match\_level'].value\_counts().to\_dict(),

'top\_skills': {},

'common\_gaps': {},

'recommendations': []

}

# Score distribution

score\_ranges = {

'Excellent (85-100)': len(evaluations[evaluations['final\_score'] >= 85]),

'Strong (75-84)': len(evaluations[(evaluations['final\_score'] >= 75) & (evaluations['final\_score'] < 85)]),

'Good (65-74)': len(evaluations[(evaluations['final\_score'] >= 65) & (evaluations['final\_score'] < 75)]),

'Moderate (50-64)': len(evaluations[(evaluations['final\_score'] >= 50) & (evaluations['final\_score'] < 65)]),

'Weak (<50)': len(evaluations[evaluations['final\_score'] < 50])

}

insights['score\_distribution'] = score\_ranges

# Generate recommendations

if insights['avg\_match\_score'] < 60:

insights['recommendations'].append("Consider expanding sourcing channels or adjusting requirements")

if score\_ranges['Excellent (85-100)'] == 0:

insights['recommendations'].append("No excellent matches found - review job requirements")

return insights

# ----- Visualization Components -----

class Visualizer:

"""Enhanced visualization components"""

@staticmethod

def create\_candidate\_radar(scores: Dict[str, float], name: str):

"""Create radar chart for candidate profile"""

if not go:

st.info("Install plotly for visualizations")

return None

categories = list(scores.keys())

values = list(scores.values())

fig = go.Figure()

fig.add\_trace(go.Scatterpolar(

r=values,

theta=categories,

fill='toself',

name=name,

line=dict(color='rgb(37, 99, 235)', width=2),

fillcolor='rgba(37, 99, 235, 0.2)'

))

fig.update\_layout(

polar=dict(

radialaxis=dict(

visible=True,

range=[0, 100],

tickfont=dict(size=10)

),

angularaxis=dict(

tickfont=dict(size=12)

)

),

showlegend=False,

height=400,

margin=dict(l=80, r=80, t=40, b=40)

)

return fig

@staticmethod

def create\_skill\_heatmap(candidates: List[str], skills: List[str], scores: Dict[str, Dict[str, float]]):

"""Create skill heatmap for batch comparison"""

if not go or not pd:

return None

# Prepare data

data = []

for candidate in candidates:

row = []

for skill in skills:

score = scores.get(candidate, {}).get(skill, 0)

row.append(score)

data.append(row)

fig = go.Figure(data=go.Heatmap(

z=data,

x=skills,

y=candidates,

colorscale='Blues',

text=data,

texttemplate='%{text:.0f}',

textfont={"size": 10}

))

fig.update\_layout(

title="Skills Competency Heatmap",

height=400 + len(candidates) \* 30,

xaxis\_title="Skills",

yaxis\_title="Candidates"

)

return fig

@staticmethod

def create\_pipeline\_funnel(stage\_counts: Dict[str, int]):

"""Create recruitment pipeline funnel"""

if not go:

return None

stages = list(stage\_counts.keys())

counts = list(stage\_counts.values())

fig = go.Figure(go.Funnel(

y=stages,

x=counts,

textposition="inside",

textinfo="value+percent initial",

opacity=0.8,

marker=dict(

color=["rgb(37, 99, 235)", "rgb(59, 130, 246)",

"rgb(96, 165, 250)", "rgb(147, 197, 253)"]

)

))

fig.update\_layout(

title="Recruitment Pipeline",

height=400,

margin=dict(l=100, r=50, t=60, b=50)

)

return fig

# ----- Main Application UI -----

class TalentIQApp:

"""Main application controller"""

def \_\_init\_\_(self):

self.init\_session\_state()

self.processor = TextProcessor()

self.engine = MatchingEngine()

self.interviewer = InterviewIntelligence()

self.analytics = AnalyticsEngine()

self.visualizer = Visualizer()

def init\_session\_state(self):

"""Initialize session state variables"""

if 'current\_job\_id' not in st.session\_state:

st.session\_state.current\_job\_id = None

if 'evaluation\_results' not in st.session\_state:

st.session\_state.evaluation\_results = []

if 'theme' not in st.session\_state:

st.session\_state.theme = 'light'

def render\_header(self):

"""Render application header with branding"""

col1, col2, col3 = st.columns([1, 3, 1])

with col2:

st.markdown("""

<div style="text-align: center; padding: 20px;">

<h1 style="color: #2563eb; font-size: 2.5rem; font-weight: bold;">

🚀 TalentIQ™

</h1>

<p style="color: #64748b; font-size: 1.1rem;">

Advanced Recruitment Intelligence Platform

</p>

</div>

""", unsafe\_allow\_html=True)

with col3:

if st.button("🌙" if st.session\_state.theme == 'light' else "☀️"):

st.session\_state.theme = 'dark' if st.session\_state.theme == 'light' else 'light'

st.rerun()

def render\_sidebar(self):

"""Render sidebar with navigation and stats"""

with st.sidebar:

st.markdown("### 📊 Quick Stats")

# Get stats from database

cur = db.conn.cursor()

total\_jobs = cur.execute("SELECT COUNT(\*) FROM job\_descriptions WHERE status='active'").fetchone()[0]

total\_candidates = cur.execute("SELECT COUNT(\*) FROM candidate\_evaluations").fetchone()[0]

avg\_match = cur.execute("SELECT AVG(final\_score) FROM candidate\_evaluations").fetchone()[0] or 0

col1, col2 = st.columns(2)

col1.metric("Active Jobs", total\_jobs)

col2.metric("Candidates", total\_candidates)

st.metric("Avg Match Score", f"{avg\_match:.1f}%")

st.markdown("---")

st.markdown("### 🔧 Configuration")

# Scoring weights

st.markdown("#### Scoring Weights")

technical\_weight = st.slider("Technical Skills", 0.0, 1.0, 0.4, 0.05, key="tech\_weight")

semantic\_weight = st.slider("Semantic Match", 0.0, 1.0, 0.3, 0.05, key="sem\_weight")

experience\_weight = st.slider("Experience", 0.0, 1.0, 0.2, 0.05, key="exp\_weight")

cultural\_weight = st.slider("Cultural Fit", 0.0, 1.0, 0.1, 0.05, key="cult\_weight")

# Normalize weights

total\_weight = technical\_weight + semantic\_weight + experience\_weight + cultural\_weight

if total\_weight > 0:

st.session\_state.weights = {

'technical': technical\_weight / total\_weight,

'semantic': semantic\_weight / total\_weight,

'experience': experience\_weight / total\_weight,

'cultural': cultural\_weight / total\_weight

}

st.markdown("---")

st.markdown("### 💡 Tips")

st.info("""

• Upload multiple resumes for batch processing

• Use semantic matching for better results

• Export reports for offline review

""")

def job\_management\_tab(self):

"""Job description management interface"""

st.markdown("## 📋 Job Management")

tab1, tab2, tab3 = st.tabs(["➕ Create New Job", "📂 Manage Existing", "📊 Job Analytics"])

with tab1:

self.create\_job\_form()

with tab2:

self.manage\_jobs()

with tab3:

self.job\_analytics()

def create\_job\_form(self):

"""Create new job posting form"""

st.markdown("### Create New Job Posting")

col1, col2 = st.columns(2)

with col1:

title = st.text\_input("Job Title \*", placeholder="e.g., Senior Software Engineer")

department = st.text\_input("Department", placeholder="e.g., Engineering")

company = st.text\_input("Company", placeholder="e.g., TechCorp Inc.")

location = st.text\_input("Location", placeholder="e.g., San Francisco, CA")

with col2:

employment\_type = st.selectbox("Employment Type",

["Full-time", "Part-time", "Contract", "Internship"])

salary\_range = st.text\_input("Salary Range", placeholder="e.g., $120k-$180k")

exp\_min = st.number\_input("Min Experience (years)", min\_value=0, max\_value=20, value=2)

exp\_max = st.number\_input("Max Experience (years)", min\_value=0, max\_value=30, value=5)

# JD upload or paste

st.markdown("### Job Description")

upload\_option = st.radio("Choose input method:", ["Upload File", "Paste Text"], horizontal=True)

jd\_text = ""

if upload\_option == "Upload File":

jd\_file = st.file\_uploader("Upload JD", type=['pdf', 'docx', 'txt'])

if jd\_file:

jd\_text = self.processor.extract\_text\_from\_file(jd\_file)

else:

jd\_text = st.text\_area("Paste Job Description", height=200)

# Skills extraction

col1, col2 = st.columns(2)

with col1:

must\_have = st.text\_area("Must-Have Skills (comma-separated)",

placeholder="Python, Django, PostgreSQL, AWS")

with col2:

good\_to\_have = st.text\_area("Good-to-Have Skills (comma-separated)",

placeholder="Docker, Kubernetes, React, CI/CD")

# Auto-extract skills button

if jd\_text and st.button("🤖 Auto-Extract Skills from JD"):

extracted\_skills = self.processor.extract\_skills(jd\_text)

all\_skills = []

for category, skills in extracted\_skills.items():

all\_skills.extend(skills)

# Suggest must-have and good-to-have

if all\_skills:

mid = len(all\_skills) // 2

st.info(f"Found {len(all\_skills)} skills. Suggested distribution:")

st.write("\*\*Must-Have:\*\* " + ", ".join(all\_skills[:mid]))

st.write("\*\*Good-to-Have:\*\* " + ", ".join(all\_skills[mid:]))

# Save button

if st.button("💾 Save Job Posting", type="primary"):

if not title or not jd\_text:

st.error("Job title and description are required!")

else:

# Parse skills

must\_have\_list = [s.strip() for s in must\_have.split(",") if s.strip()]

good\_to\_have\_list = [s.strip() for s in good\_to\_have.split(",") if s.strip()]

# Save to database

cur = db.conn.cursor()

metadata = json.dumps({

'created\_by': 'admin',

'version': 1

})

cur.execute("""

INSERT INTO job\_descriptions

(title, department, company, location, employment\_type, salary\_range,

raw\_text, must\_have\_skills, good\_to\_have\_skills, experience\_min,

experience\_max, metadata, status)

VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)

""", (title, department, company, location, employment\_type, salary\_range,

jd\_text, json.dumps(must\_have\_list), json.dumps(good\_to\_have\_list),

exp\_min, exp\_max, metadata, 'active'))

db.conn.commit()

job\_id = cur.lastrowid

st.session\_state.current\_job\_id = job\_id

st.success(f"✅ Job posting created successfully! (ID: {job\_id})")

st.balloons()

def manage\_jobs(self):

"""Manage existing job postings"""

st.markdown("### Existing Job Postings")

# Load jobs from database

query = """

SELECT id, title, department, location, status,

(SELECT COUNT(\*) FROM candidate\_evaluations WHERE job\_id = jd.id) as candidates,

created\_at

FROM job\_descriptions jd

ORDER BY created\_at DESC

"""

if pd:

df = pd.read\_sql\_query(query, db.conn)

if not df.empty:

# Add action column

df['Actions'] = df['id'].apply(lambda x: f"Job\_{x}")

# Display with filters

status\_filter = st.selectbox("Filter by Status", ["All", "active", "archived"])

if status\_filter != "All":

df = df[df['status'] == status\_filter]

# Format dates

df['created\_at'] = pd.to\_datetime(df['created\_at']).dt.strftime('%Y-%m-%d')

# Display table

st.dataframe(

df[['id', 'title', 'department', 'location', 'candidates', 'status', 'created\_at']],

use\_container\_width=True,

hide\_index=True

)

# Action buttons

col1, col2, col3 = st.columns(3)

with col1:

job\_id = st.selectbox("Select Job ID", df['id'].tolist())

with col2:

if st.button("📝 Select for Evaluation"):

st.session\_state.current\_job\_id = job\_id

st.success(f"Selected Job ID: {job\_id}")

with col3:

if st.button("📊 View Analytics"):

st.session\_state.view\_job\_analytics = job\_id

else:

st.info("No job postings found. Create one to get started!")

else:

st.warning("Install pandas for better data management")

def job\_analytics(self):

"""Display job-specific analytics"""

st.markdown("### Job Performance Analytics")

if not st.session\_state.current\_job\_id:

st.info("Select a job first to view analytics")

return

job\_id = st.session\_state.current\_job\_id

# Get job details

cur = db.conn.cursor()

job = cur.execute("SELECT \* FROM job\_descriptions WHERE id = ?", (job\_id,)).fetchone()

if not job:

st.error("Job not found")

return

st.markdown(f"#### Analytics for: {job[1]}")

# Get evaluation metrics

metrics = cur.execute("""

SELECT

COUNT(\*) as total\_candidates,

AVG(final\_score) as avg\_score,

MAX(final\_score) as max\_score,

MIN(final\_score) as min\_score,

COUNT(CASE WHEN final\_score >= 75 THEN 1 END) as strong\_matches

FROM candidate\_evaluations

WHERE job\_id = ?

""", (job\_id,)).fetchone()

if metrics[0] > 0:

# Display metrics

col1, col2, col3, col4 = st.columns(4)

col1.metric("Total Candidates", metrics[0])

col2.metric("Avg Match Score", f"{metrics[1]:.1f}%")

col3.metric("Best Match", f"{metrics[2]:.1f}%")

col4.metric("Strong Matches", metrics[4])

# Score distribution chart

if pd and px:

evals = pd.read\_sql\_query(

"SELECT final\_score, candidate\_name FROM candidate\_evaluations WHERE job\_id = ?",

db.conn, params=(job\_id,)

)

if not evals.empty:

fig = px.histogram(

evals, x='final\_score', nbins=20,

title="Score Distribution",

labels={'final\_score': 'Match Score (%)', 'count': 'Number of Candidates'}

)

fig.update\_traces(marker\_color='rgb(37, 99, 235)')

st.plotly\_chart(fig, use\_container\_width=True)

# Top candidates

st.markdown("#### 🏆 Top 5 Candidates")

top5 = evals.nlargest(5, 'final\_score')

for idx, row in top5.iterrows():

st.write(f"\*\*{row['candidate\_name']}\*\* - Match Score: {row['final\_score']:.1f}%")

else:

st.info("No evaluations yet for this job posting")

def evaluation\_tab(self):

"""Candidate evaluation interface"""

st.markdown("## 🎯 Candidate Evaluation")

if not st.session\_state.current\_job\_id:

st.warning("⚠️ Please select a job posting first from the Job Management tab")

return

# Get current job details

cur = db.conn.cursor()

job = cur.execute("SELECT \* FROM job\_descriptions WHERE id = ?",

(st.session\_state.current\_job\_id,)).fetchone()

st.info(f"📌 Evaluating for: \*\*{job[1]}\*\* | {job[3]} | {job[4]}")

# Resume upload

st.markdown("### Upload Resumes")

resumes = st.file\_uploader(

"Upload candidate resumes (PDF, DOCX, TXT)",

type=['pdf', 'docx', 'txt'],

accept\_multiple\_files=True,

help="You can select multiple files for batch processing"

)

# Advanced options

with st.expander("⚙️ Advanced Options"):

col1, col2 = st.columns(2)

with col1:

include\_cultural = st.checkbox("Include Cultural Fit Analysis", value=True)

generate\_questions = st.checkbox("Generate Interview Questions", value=True)

with col2:

export\_reports = st.checkbox("Generate PDF Reports", value=False)

anonymous\_mode = st.checkbox("Anonymous Evaluation Mode", value=False)

# Evaluation button

if st.button("🚀 Start Evaluation", type="primary", disabled=not resumes):

self.process\_evaluations(job, resumes, include\_cultural, generate\_questions,

export\_reports, anonymous\_mode)

def process\_evaluations(self, job, resumes, include\_cultural, generate\_questions,

export\_reports, anonymous\_mode):

"""Process batch resume evaluations"""

# Parse job requirements

must\_have = json.loads(job[8]) if job[8] else []

good\_to\_have = json.loads(job[9]) if job[9] else []

all\_skills = must\_have + good\_to\_have

results = []

progress = st.progress(0)

status = st.empty()

for idx, resume\_file in enumerate(resumes):

status.text(f"Processing {idx+1}/{len(resumes)}: {resume\_file.name}")

# Extract resume text

resume\_text = self.processor.extract\_text\_from\_file(resume\_file)

cleaned\_text = self.processor.clean\_and\_normalize(resume\_text)

# Extract candidate info

contact = self.processor.extract\_contact\_info(cleaned\_text)

experience = self.processor.extract\_experience\_years(cleaned\_text)

skills = self.processor.extract\_skills(cleaned\_text)

# Flatten skills

candidate\_skills = []

for category, skill\_list in skills.items():

candidate\_skills.extend(skill\_list)

# Calculate scores

tech\_score, matched, gaps = self.engine.calculate\_technical\_match(

all\_skills, candidate\_skills

)

semantic\_score = self.engine.calculate\_semantic\_similarity(

job[7], cleaned\_text

)

exp\_score = self.engine.calculate\_experience\_match(

(job[10], job[11]), experience

)

cultural\_score = 0

if include\_cultural:

cultural\_score = self.engine.calculate\_cultural\_fit(cleaned\_text)

# Calculate weighted final score

weights = st.session\_state.get('weights', {

'technical': 0.4,

'semantic': 0.3,

'experience': 0.2,

'cultural': 0.1

})

final\_score = (

tech\_score \* weights['technical'] +

semantic\_score \* weights['semantic'] +

exp\_score \* weights['experience'] +

cultural\_score \* weights['cultural']

)

match\_level = self.engine.determine\_match\_level(final\_score)

# Generate recommendations

recommendations = []

if final\_score >= 75:

recommendations.append("Fast-track to technical interview")

elif final\_score >= 60:

recommendations.append("Schedule screening call")

else:

recommendations.append("Review with hiring manager")

# Generate interview questions

questions = []

if generate\_questions:

questions = self.interviewer.generate\_questions(gaps, matched, "mid")

# Risk factors

risk\_factors = self.engine.generate\_risk\_factors(

CandidateProfile(

name=resume\_file.name if anonymous\_mode else contact.get('email', resume\_file.name),

email=contact.get('email', ''),

phone=contact.get('phone', ''),

resume\_text=cleaned\_text,

skills=candidate\_skills,

experience\_years=experience,

education=[],

certifications=[]

),

JobDescription(

id=job[0], title=job[1], department=job[2], raw\_text=job[7],

must\_have\_skills=must\_have, good\_to\_have\_skills=good\_to\_have,

experience\_range=(job[10], job[11]), created\_at=datetime.now(),

metadata={}

),

gaps

)

# Save to database

candidate\_name = f"Candidate\_{idx+1}" if anonymous\_mode else resume\_file.name

cur = db.conn.cursor()

cur.execute("""

INSERT INTO candidate\_evaluations

(job\_id, candidate\_name, candidate\_email, candidate\_phone, resume\_hash,

resume\_text, technical\_score, cultural\_fit\_score, experience\_score,

semantic\_score, final\_score, match\_level, strengths, gaps,

recommendations, interview\_questions, risk\_factors, status)

VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)

""", (

job[0], candidate\_name, contact.get('email', ''), contact.get('phone', ''),

hashlib.md5(cleaned\_text.encode()).hexdigest(), cleaned\_text[:5000],

tech\_score, cultural\_score, exp\_score, semantic\_score, final\_score,

match\_level.value, json.dumps(matched[:10]), json.dumps(gaps[:10]),

json.dumps(recommendations), json.dumps(questions),

json.dumps(risk\_factors), 'evaluated'

))

db.conn.commit()

results.append({

'name': candidate\_name,

'scores': {

'Technical': tech\_score,

'Semantic': semantic\_score,

'Experience': exp\_score,

'Cultural': cultural\_score

},

'final\_score': final\_score,

'match\_level': match\_level,

'matched': matched,

'gaps': gaps,

'recommendations': recommendations,

'questions': questions,

'risks': risk\_factors

})

progress.progress((idx + 1) / len(resumes))

status.empty()

progress.empty()

# Display results

self.display\_evaluation\_results(results, export\_reports)

def display\_evaluation\_results(self, results, export\_reports):

"""Display evaluation results with visualizations"""

st.success(f"✅ Successfully evaluated {len(results)} candidates!")

# Sort by final score

results.sort(key=lambda x: x['final\_score'], reverse=True)

# Summary metrics

st.markdown("### 📊 Evaluation Summary")

col1, col2, col3, col4 = st.columns(4)

avg\_score = sum(r['final\_score'] for r in results) / len(results)

excellent = sum(1 for r in results if r['match\_level'] == MatchLevel.EXCELLENT)

strong = sum(1 for r in results if r['match\_level'] == MatchLevel.STRONG)

col1.metric("Average Match", f"{avg\_score:.1f}%")

col2.metric("Excellent Matches", excellent)

col3.metric("Strong Matches", strong)

col4.metric("Total Evaluated", len(results))

# Top candidates

st.markdown("### 🏆 Top Candidates")

for idx, candidate in enumerate(results[:5], 1):

with st.expander(f"{idx}. {candidate['name']} - {candidate['final\_score']:.1f}% ({candidate['match\_level'].value})"):

# Scores radar chart

col1, col2 = st.columns([2, 3])

with col1:

st.markdown("#### Match Scores")

for metric, score in candidate['scores'].items():

st.metric(metric, f"{score:.1f}%")

with col2:

fig = self.visualizer.create\_candidate\_radar(

candidate['scores'],

candidate['name']

)

if fig:

st.plotly\_chart(fig, use\_container\_width=True)

# Strengths and gaps

col1, col2 = st.columns(2)

with col1:

st.markdown("#### ✅ Strengths")

for skill in candidate['matched'][:5]:

st.write(f"• {skill}")

with col2:

st.markdown("#### ⚠️ Gaps")

for gap in candidate['gaps'][:5]:

st.write(f"• {gap}")

# Recommendations

st.markdown("#### 💡 Recommendations")

for rec in candidate['recommendations']:

st.info(rec)

# Interview questions

if candidate['questions']:

st.markdown("#### 🎤 Suggested Interview Questions")

for q in candidate['questions'][:5]:

st.write(f"• {q}")

# Risk factors

if candidate['risks']:

st.markdown("#### ⚠️ Risk Factors")

for risk in candidate['risks']:

st.warning(risk)

# Export button

if export\_reports:

if st.button(f"📄 Export Report for {candidate['name']}", key=f"export\_{idx}"):

st.info("Report export feature coming soon!")

# Batch comparison

if len(results) > 1:

st.markdown("### 🔄 Candidate Comparison")

# Prepare data for heatmap

candidates = [r['name'] for r in results[:10]]

skills = list(set(skill for r in results for skill in r['matched']))[:15]

skill\_scores = {}

for r in results[:10]:

skill\_scores[r['name']] = {}

for skill in skills:

skill\_scores[r['name']][skill] = 100 if skill in r['matched'] else 0

fig = self.visualizer.create\_skill\_heatmap(candidates, skills, skill\_scores)

if fig:

st.plotly\_chart(fig, use\_container\_width=True)

# Export all results

if pd and results:

df\_results = pd.DataFrame([{

'Candidate': r['name'],

'Final Score': r['final\_score'],

'Match Level': r['match\_level'].value,

'Technical': r['scores']['Technical'],

'Semantic': r['scores']['Semantic'],

'Experience': r['scores']['Experience'],

'Cultural': r['scores']['Cultural']

} for r in results])

csv = df\_results.to\_csv(index=False)

st.download\_button(

"📥 Download Results (CSV)",

csv,

"evaluation\_results.csv",

"text/csv",

key='download-csv'

)

def analytics\_dashboard\_tab(self):

"""Analytics and insights dashboard"""

st.markdown("## 📈 Analytics Dashboard")

tab1, tab2, tab3, tab4 = st.tabs(["Overview", "Talent Pool", "Pipeline", "Insights"])

with tab1:

self.render\_overview\_analytics()

with tab2:

self.render\_talent\_pool\_analytics()

with tab3:

self.render\_pipeline\_analytics()

with tab4:

self.render\_insights()

def render\_overview\_analytics(self):

"""Render overview analytics"""

st.markdown("### 📊 Recruitment Overview")

if not pd:

st.warning("Install pandas for analytics features")

return

# Date range filter

col1, col2 = st.columns(2)

with col1:

start\_date = st.date\_input("Start Date", datetime.now() - timedelta(days=30))

with col2:

end\_date = st.date\_input("End Date", datetime.now())

# Load data

query = """

SELECT

DATE(created\_at) as date,

COUNT(\*) as evaluations,

AVG(final\_score) as avg\_score

FROM candidate\_evaluations

WHERE DATE(created\_at) BETWEEN ? AND ?

GROUP BY DATE(created\_at)

ORDER BY date

"""

df = pd.read\_sql\_query(query, db.conn, params=(start\_date, end\_date))

if not df.empty:

# Time series chart

if px:

fig = px.line(df, x='date', y='evaluations',

title="Daily Evaluation Trend",

labels={'evaluations': 'Number of Evaluations', 'date': 'Date'})

fig.update\_traces(line\_color='rgb(37, 99, 235)')

st.plotly\_chart(fig, use\_container\_width=True)

# Score trend

fig2 = px.line(df, x='date', y='avg\_score',

title="Average Match Score Trend",

labels={'avg\_score': 'Average Score (%)', 'date': 'Date'})

fig2.update\_traces(line\_color='rgb(16, 185, 129)')

st.plotly\_chart(fig2, use\_container\_width=True)

else:

st.info("No data available for selected date range")

def render\_talent\_pool\_analytics(self):

"""Render talent pool analytics"""

st.markdown("### 👥 Talent Pool Analysis")

if not pd:

return

# Load candidate data

query = """

SELECT

match\_level,

COUNT(\*) as count,

AVG(technical\_score) as avg\_technical,

AVG(semantic\_score) as avg\_semantic,

AVG(experience\_score) as avg\_experience

FROM candidate\_evaluations

GROUP BY match\_level

"""

df = pd.read\_sql\_query(query, db.conn)

if not df.empty and px:

# Match level distribution

fig = px.pie(df, values='count', names='match\_level',

title="Talent Distribution by Match Level")

st.plotly\_chart(fig, use\_container\_width=True)

# Score comparison

df\_melted = df.melt(id\_vars=['match\_level'],

value\_vars=['avg\_technical', 'avg\_semantic', 'avg\_experience'],

var\_name='Score Type', value\_name='Score')

fig2 = px.bar(df\_melted, x='match\_level', y='Score', color='Score Type',

title="Average Scores by Match Level",

barmode='group')

st.plotly\_chart(fig2, use\_container\_width=True)

def render\_pipeline\_analytics(self):

"""Render recruitment pipeline analytics"""

st.markdown("### 🔄 Recruitment Pipeline")

# Simulated pipeline data (you can replace with actual data)

pipeline\_data = {

'Applied': 500,

'Screened': 300,

'Technical Interview': 100,

'HR Interview': 50,

'Offered': 20,

'Accepted': 15

}

fig = self.visualizer.create\_pipeline\_funnel(pipeline\_data)

if fig:

st.plotly\_chart(fig, use\_container\_width=True)

# Conversion metrics

st.markdown("### Conversion Rates")

col1, col2, col3 = st.columns(3)

col1.metric("Screen to Interview", "33.3%", "↑ 5%")

col2.metric("Interview to Offer", "20%", "↓ 2%")

col3.metric("Offer Acceptance", "75%", "→ 0%")

def render\_insights(self):

"""Render AI-generated insights"""

st.markdown("### 🤖 AI-Powered Insights")

if not pd:

return

# Load recent evaluation data

df = pd.read\_sql\_query("""

SELECT \* FROM candidate\_evaluations

ORDER BY created\_at DESC

""", db.conn)

if not df.empty:

# Generate insights

insights = self.analytics.generate\_batch\_insights(df)

# Display key insights

if insights:

# Score distribution insight

st.markdown("#### 📊 Score Distribution")

for range\_name, count in insights.get('score\_distribution', {}).items():

if count > 0:

st.write(f"• {range\_name}: \*\*{count}\*\* candidates")

# Recommendations

if insights.get('recommendations'):

st.markdown("#### 💡 Recommendations")

for rec in insights['recommendations']:

st.info(rec)

# Talent gap analysis

st.markdown("#### 🎯 Talent Gap Analysis")

gaps\_query = """

SELECT gaps, COUNT(\*) as frequency

FROM candidate\_evaluations

WHERE gaps IS NOT NULL AND gaps != '[]'

GROUP BY gaps

ORDER BY frequency DESC

LIMIT 5

"""

gaps\_df = pd.read\_sql\_query(gaps\_query, db.conn)

if not gaps\_df.empty:

st.write("\*\*Most Common Skill Gaps:\*\*")

for \_, row in gaps\_df.iterrows():

try:

gap\_list = json.loads(row['gaps'])

if gap\_list:

st.write(f"• {', '.join(gap\_list[:3])}")

except:

continue

# Success factors

st.markdown("#### ✨ Success Factors")

top\_performers = df[df['final\_score'] >= 80]

if not top\_performers.empty:

st.write(f"\*\*{len(top\_performers)}\*\* candidates scored above 80%")

st.write("Common attributes of top performers:")

st.write("• Strong technical skill alignment")

st.write("• Relevant industry experience")

st.write("• Cultural fit indicators present")

else:

st.info("No evaluation data available for insights generation")

def settings\_tab(self):

"""Settings and configuration interface"""

st.markdown("## ⚙️ Settings & Configuration")

tab1, tab2, tab3 = st.tabs(["General Settings", "Export/Import", "About"])

with tab1:

st.markdown("### General Settings")

# Evaluation settings

st.markdown("#### Evaluation Configuration")

col1, col2 = st.columns(2)

with col1:

min\_match\_threshold = st.slider(

"Minimum Match Threshold (%)",

min\_value=0,

max\_value=100,

value=50,

help="Candidates below this threshold will be flagged"

)

auto\_reject = st.checkbox(

"Auto-reject below threshold",

value=False,

help="Automatically reject candidates below minimum threshold"

)

with col2:

batch\_size = st.number\_input(

"Max Batch Size",

min\_value=1,

max\_value=100,

value=20,

help="Maximum number of resumes to process at once"

)

enable\_notifications = st.checkbox(

"Enable Email Notifications",

value=False,

help="Send email notifications for high-match candidates"

)

# API Settings

st.markdown("#### API Configuration")

api\_key = st.text\_input("API Key (for advanced features)", type="password")

webhook\_url = st.text\_input("Webhook URL (optional)", placeholder="https://your-webhook.com/endpoint")

if st.button("💾 Save Settings"):

st.success("Settings saved successfully!")

with tab2:

st.markdown("### Data Export/Import")

col1, col2 = st.columns(2)

with col1:

st.markdown("#### Export Data")

export\_type = st.selectbox("Select data to export",

["All Evaluations", "Job Postings", "Analytics Report"])

if st.button("📥 Export"):

if pd:

if export\_type == "All Evaluations":

df = pd.read\_sql\_query("SELECT \* FROM candidate\_evaluations", db.conn)

elif export\_type == "Job Postings":

df = pd.read\_sql\_query("SELECT \* FROM job\_descriptions", db.conn)

else:

df = pd.read\_sql\_query("""

SELECT j.title, COUNT(c.id) as candidates,

AVG(c.final\_score) as avg\_score

FROM job\_descriptions j

LEFT JOIN candidate\_evaluations c ON j.id = c.job\_id

GROUP BY j.id

""", db.conn)

csv = df.to\_csv(index=False)

st.download\_button(

f"Download {export\_type}",

csv,

f"{export\_type.lower().replace(' ', '\_')}.csv",

"text/csv"

)

with col2:

st.markdown("#### Import Data")

import\_file = st.file\_uploader("Upload CSV to import", type=['csv'])

if import\_file:

st.info("Import functionality coming soon!")

# Database management

st.markdown("#### Database Management")

col1, col2, col3 = st.columns(3)

with col1:

if st.button("🔄 Backup Database"):

st.success("Database backed up successfully!")

with col2:

if st.button("🗑️ Clear Old Data"):

st.warning("This will remove evaluations older than 6 months")

with col3:

if st.button("📊 Optimize Database"):

cur = db.conn.cursor()

cur.execute("VACUUM")

db.conn.commit()

st.success("Database optimized!")

with tab3:

st.markdown("### About TalentIQ™")

st.markdown("""

#### Version 2.0.0

TalentIQ is an advanced AI-powered recruitment intelligence platform designed to

streamline and optimize your hiring process.

\*\*Key Features:\*\*

- 🤖 AI-powered resume screening and matching

- 📊 Advanced analytics and insights

- 🎯 Multi-dimensional candidate evaluation

- 📈 Real-time recruitment metrics

- 🔄 Batch processing capabilities

- 📱 Modern, responsive interface

\*\*Technology Stack:\*\*

- Python 3.8+

- Streamlit

- SQLite Database

- Machine Learning Models

- Natural Language Processing

\*\*Support:\*\*

- Documentation: [docs.talentiq.ai](https://docs.talentiq.ai)

- Email: support@talentiq.ai

- Version: 2.0.0

- Last Updated: 2024

---

© 2024 TalentIQ™. All rights reserved.

""")

# System info

st.markdown("#### System Information")

col1, col2 = st.columns(2)

with col1:

st.metric("Database Size", "12.5 MB")

st.metric("Total Evaluations",

db.conn.execute("SELECT COUNT(\*) FROM candidate\_evaluations").fetchone()[0])

with col2:

st.metric("Active Jobs",

db.conn.execute("SELECT COUNT(\*) FROM job\_descriptions WHERE status='active'").fetchone()[0])

st.metric("Avg Processing Time", "2.3 sec/resume")

def run(self):

"""Main application entry point"""

# Apply custom CSS

st.markdown("""

<style>

.stTabs [data-baseweb="tab-list"] {

gap: 24px;

}

.stTabs [data-baseweb="tab"] {

padding-left: 20px;

padding-right: 20px;

height: 50px;

white-space: pre-wrap;

background-color: transparent;

border-radius: 8px;

font-weight: 500;

}

.stTabs [aria-selected="true"] {

background-color: #2563eb;

color: white;

}

div[data-testid="metric-container"] {

background-color: #f8fafc;

border: 1px solid #e2e8f0;

padding: 15px;

border-radius: 8px;

box-shadow: 0 1px 3px rgba(0,0,0,0.05);

}

.stButton > button {

border-radius: 8px;

font-weight: 500;

transition: all 0.3s;

}

.stButton > button:hover {

transform: translateY(-2px);

box-shadow: 0 4px 12px rgba(0,0,0,0.15);

}

</style>

""", unsafe\_allow\_html=True)

# Render header

self.render\_header()

# Render sidebar

self.render\_sidebar()

# Main navigation

tab1, tab2, tab3, tab4 = st.tabs([

"📋 Job Management",

"🎯 Candidate Evaluation",

"📈 Analytics Dashboard",

"⚙️ Settings"

])

with tab1:

self.job\_management\_tab()

with tab2:

self.evaluation\_tab()

with tab3:

self.analytics\_dashboard\_tab()

with tab4:

self.settings\_tab()

# Footer

st.markdown("---")

st.markdown(

"""

<div style="text-align: center; color: #64748b; padding: 20px;">

<p>Team Ghost</p>

</div>

""",

unsafe\_allow\_html=True

)

# ----- Application Entry Point -----

if \_\_name\_\_ == "\_\_main\_\_":

app = TalentIQApp()

app.run()