**Cursor AI Prompts for Resume AI Feature**

**Context for All Prompts**

**Project Background:** Mingus is a personal finance assistant targeting African American professionals aged 25-35 earning $40k-100k. This resume AI feature helps users find higher-paying jobs by analyzing their resume and matching them with opportunities that offer 15-30% salary increases. The target markets are in Atlanta, Houston, DC Metro, Dallas, New York, Philadelphia, Chicago, Charlotte, Miami, and Baltimore.

**Tech Stack:** Python, Flask, PyPDF2, python-docx, basic AI/ML, free APIs **Budget:** Ultra-low cost MVP using free tiers **User Skill Level:** Basic Python knowledge

**Prompt 1: Project Setup and Structure**

Create a complete Python project structure for a resume AI analysis tool called "mingus-resume-ai".

Requirements:

- Flask web application for resume upload and job matching

- Resume parsing for PDF and DOCX files using PyPDF2 and python-docx

- Integration with free job search APIs

- Target users: African American professionals aged 25-35, $40k-100k salary

- Focus on skills extraction and job matching for career advancement

Create:

1. requirements.txt with exact versions for stability

2. .env.example file for environment variables

3. config.py for application configuration

4. Project folder structure with appropriate directories

5. .gitignore file for Python projects

6. README.md with setup instructions

Include proper imports and dependencies for:

- Flask web framework

- PDF/DOCX parsing (PyPDF2, python-docx)

- HTTP requests for job APIs

- Environment variable management

- File upload handling

**Prompt 2: Resume Parser Core Component**

Create a comprehensive ResumeParser class for extracting structured data from PDF and DOCX resume files.

Context: This is for Mingus, a finance app helping African American professionals (ages 25-35, $40k-100k income) find better-paying jobs. The parser needs to extract skills relevant to career advancement.

Requirements:

1. Support PDF and DOCX file formats

2. Extract: name, email, phone, work experience, education, skills

3. Skills should focus on: technical skills (Python, JavaScript, SQL, Excel), business skills (project management, sales, marketing), and soft skills (leadership, communication)

4. Handle common resume formatting variations

5. Include error handling and logging

6. Return structured dictionary format

7. Skills extraction should be comprehensive for job matching

Key features:

- Robust text extraction from both file types

- Pattern matching for contact information

- Section detection for experience/education

- Skills matching against predefined keywords relevant to target demographic

- Clean, structured output for job matching algorithm

Include detailed comments and example usage. Make it production-ready with proper error handling.

**Prompt 3: Job Search API Integration**

Create a JobSearcher class that integrates with free job search APIs to find opportunities for career advancement.

Context: Mingus users are African American professionals earning $40k-100k who need 15-30% salary increases. Target locations: Atlanta, Houston, DC Metro, Dallas, New York, Philadelphia, Chicago, Charlotte, Miami, Baltimore.

Requirements:

1. Primary integration with Adzuna API (free tier: 1000 searches/month)

2. Search jobs based on extracted resume skills

3. Filter by location (target metro areas)

4. Filter by salary minimum (15%+ increase from current)

5. Return structured job data: title, company, location, salary, description, URL

6. Handle API rate limiting and errors gracefully

7. Support skills-based job searches

8. Include backup search strategies

Features needed:

- Job class/dataclass for structured job data

- Search by individual skills and skill combinations

- Location-aware searching for target markets

- Salary-focused filtering for career advancement

- Deduplication of job results

- Professional formatting of salary information

Include configuration for API keys, rate limiting, and error handling. Make it easily extensible for additional job APIs later.

**Prompt 4: AI Matching Algorithm**

Create a SimpleAIMatcher class that scores and ranks job opportunities based on resume analysis.

Context: Mingus helps African American professionals (25-35, $40k-100k) find career advancement opportunities. The matcher should prioritize jobs offering 15-30% salary increases and good skills alignment.

Requirements:

1. Calculate match scores between resume data and job descriptions

2. Weighted scoring: skills match (40%), experience level (30%), salary increase potential (30%)

3. Generate human-readable explanations for why jobs match

4. Rank jobs by overall match score

5. Identify salary improvement potential

6. Highlight matching skills between resume and job

7. Handle edge cases (missing salary data, unclear job descriptions)

Key features:

- JobMatch dataclass with job, score, salary increase, matching skills, reasons

- Skills matching algorithm with relevance weighting

- Experience level assessment (entry/mid/senior)

- Salary increase calculation and formatting

- Match reasoning generation for user understanding

- Comprehensive ranking system

The algorithm should be explainable to users - they should understand why a job is recommended. Focus on actionable career advancement rather than just any job match.

Include examples and test cases for validation.

**Prompt 5: Flask Web Application**

Create a complete Flask web application for the Mingus resume AI feature with file upload, processing, and results display.

Context: Targeting African American professionals aged 25-35 earning $40k-100k. Users upload resumes to find higher-paying job opportunities. The app should feel professional and trustworthy.

Requirements:

1. File upload endpoint supporting PDF/DOCX (max 5MB)

2. Resume processing using ResumeParser class

3. Job search using JobSearcher class

4. Job ranking using SimpleAIMatcher class

5. Results page showing top job matches with explanations

6. Error handling for file processing failures

7. Security considerations (file validation, secure filenames)

8. Clean temporary file management

Features needed:

- Root route with upload form

- File upload with validation and security checks

- Resume processing pipeline (parse → search → match → rank)

- Results display with match scores and explanations

- Professional UI appropriate for target demographic

- Mobile-responsive design basics

- Progress indication for processing

- Error messages and user feedback

Include:

- Proper Flask configuration

- Security headers and file validation

- Temporary file cleanup

- User-friendly error handling

- API endpoint for potential mobile integration

- Development and production configurations

Make it deployment-ready for platforms like Heroku or PythonAnywhere.

**Prompt 6: HTML Templates and UI**

Create professional HTML templates for the Mingus resume AI web application.

Context: Mingus serves African American professionals (25-35, $40k-100k) seeking career advancement. The design should feel trustworthy, professional, and culturally relevant. Users are looking for financial improvement through better jobs.

Create templates for:

1. index.html - Landing page with value proposition and upload form

2. upload.html - Resume upload page with form and instructions

3. results.html - Job matches display with scores and explanations

4. layout.html - Base template with navigation and branding

Design requirements:

- Professional, clean design appropriate for career services

- Mobile-responsive (many users will access via phone)

- Clear value proposition focusing on salary increases

- Trust indicators (security, privacy, professional appearance)

- Progress indicators for file processing

- Accessibility considerations (WCAG guidelines)

- Color scheme that's professional and inclusive

Key features:

- Compelling headlines about career advancement and financial growth

- Clear file upload interface with drag-and-drop

- Job results cards showing match percentage, salary increase, matching skills

- Call-to-action buttons for applying to jobs

- Error messaging and loading states

- Social proof elements (testimonials, success metrics)

Include modern CSS with:

- Flexbox/Grid layouts

- Hover effects and transitions

- Professional typography

- Responsive design patterns

- Form styling and validation feedback

Make it feel like a premium career service that users would trust with their professional information.

**Prompt 7: Testing and Quality Assurance**

Create comprehensive testing setup for the Mingus resume AI feature including unit tests, integration tests, and sample data.

Context: This is a career advancement tool for African American professionals. Reliability is crucial since users are trusting us with their resume data and career decisions.

Create:

1. Unit tests for ResumeParser class (PDF/DOCX parsing, skills extraction)

2. Unit tests for JobSearcher class (API integration, search functionality)

3. Unit tests for SimpleAIMatcher class (scoring algorithm, ranking logic)

4. Integration tests for Flask application (file upload, end-to-end processing)

5. Sample test files (anonymized resume PDFs/DOCX)

6. Mock data for job search API responses

7. Performance tests for file processing

8. Error handling validation

Testing requirements:

- pytest framework setup

- Mock external API calls for reliable testing

- Test data representing target demographic (skills, experience levels)

- Edge case testing (corrupted files, missing data, API failures)

- Security testing (file upload validation, malicious file handling)

- Performance benchmarks (processing time expectations)

Quality assurance features:

- Logging setup for debugging and monitoring

- Error tracking and reporting

- Data validation throughout the pipeline

- User input sanitization

- Rate limiting testing for API integrations

Include:

- Test configuration and setup instructions

- Sample resumes with known expected outputs

- Mock job data for consistent testing

- Performance benchmarks and acceptance criteria

- CI/CD pipeline suggestions for automated testing

Focus on reliability and user trust - this tool affects people's careers and financial futures.

**Prompt 8: Deployment and Configuration**

Create deployment configuration and documentation for the Mingus resume AI feature targeting production deployment.

Context: This career advancement tool serves African American professionals seeking salary increases. It needs reliable, secure deployment with minimal ongoing costs.

Create deployment configurations for:

1. Heroku (free/hobby tier initially)

2. PythonAnywhere (alternative free option)

3. Local development setup

4. Environment variable management

5. Database setup (if needed for user tracking)

6. Monitoring and logging configuration

Requirements:

- Production-ready Flask configuration

- Environment-specific settings (dev/staging/prod)

- Security configurations (HTTPS, secure headers, file validation)

- Performance optimizations for resume processing

- Cost-effective scaling strategies

- Backup and recovery procedures for user data

- Privacy and data protection compliance

Include:

- Procfile for Heroku deployment

- WSGI configuration for production servers

- Environment variable documentation

- SSL/HTTPS setup instructions

- Database migration scripts (if applicable)

- Monitoring setup (error tracking, performance metrics)

- Cost monitoring and optimization strategies

Documentation needed:

- Step-by-step deployment guide

- Environment setup for different platforms

- Troubleshooting common deployment issues

- Scaling considerations as user base grows

- Security best practices checklist

- GDPR/privacy compliance guidelines

Focus on:

- Minimal initial costs (free tier usage)

- Easy scaling as user base grows

- Reliable performance for career-critical functionality

- Security appropriate for handling resume data

- Monitoring to ensure service reliability

**Usage Instructions for Cursor**

**How to Use These Prompts:**

1. **Copy each prompt exactly** into Cursor's chat interface
2. **Use them in order** - each builds on the previous components
3. **Customize as needed** - add specific requirements for your use case
4. **Test each component** before moving to the next prompt
5. **Iterate** - ask follow-up questions to refine the generated code

**Cursor-Specific Tips:**

* Use @workspace to reference your entire project context
* Use @filename to reference specific files when asking for modifications
* Ask Cursor to "explain this code" if you need clarification
* Request "add comments" for better documentation
* Use "optimize this code" for performance improvements

**Example Follow-up Prompts:**

@resume\_parser.py Can you add more robust error handling for corrupted PDF files?

@app.py Add rate limiting to prevent abuse of the file upload endpoint

@templates Can you make the design more accessible for screen readers?

Optimize the job matching algorithm for better performance with large numbers of jobs

Each prompt is designed to generate production-ready code that you can immediately use and deploy. Start with Prompt 1 and work through them sequentially.

**Income Comparison Feature Implementation**

**Overview**

This feature shows users how their income compares to various demographic groups, providing context and motivation for career advancement. Perfect for Mingus users seeking financial growth.

**Data Sources & APIs**

**1. U.S. Census Bureau API (FREE) ⭐ RECOMMENDED**

* **Cost**: Free with rate limits (500 requests/day)
* **Data**: American Community Survey (ACS) income data
* **Coverage**: Income by race, age, education, metro area
* **API Endpoint**: https://api.census.gov/data/2022/acs/acs1

**2. Bureau of Labor Statistics API (FREE)**

* **Cost**: Free, no registration required
* **Data**: Occupational employment and wages
* **Coverage**: Income by occupation and industry
* **API Endpoint**: https://api.bls.gov/publicAPI/v2/timeseries/data/

**3. Economic Policy Institute API**

* **Cost**: Free for basic data
* **Data**: Wage and income inequality statistics
* **Focus**: Racial and regional wage gaps

**Implementation Steps**

**Step 1: Create Income Comparison Service**

# income\_comparator.py

import requests

import pandas as pd

from typing import Dict, List, Optional

from dataclasses import dataclass

import os

@dataclass

class IncomeComparison:

user\_income: float

comparison\_group: str

median\_income: float

percentile\_rank: int

income\_gap: float

gap\_percentage: float

context\_message: str

class IncomeComparator:

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

self.census\_api\_key = os.getenv('CENSUS\_API\_KEY') # Optional but increases rate limits

self.census\_base\_url = "https://api.census.gov/data/2022/acs/acs1"

# Pre-loaded data for offline fallback (2022 ACS data)

self.fallback\_data = {

'national\_median': 70850,

'african\_american\_median': 48297,

'college\_grad\_median': 78000,

'metro\_medians': {

'Atlanta': 65000,

'Houston': 63000,

'Washington DC': 95000,

'Dallas': 67000,

'New York': 85000,

'Philadelphia': 70000,

'Chicago': 68000,

'Charlotte': 62000,

'Miami': 58000,

'Baltimore': 75000

}

}

def get\_comprehensive\_comparison(self, user\_income: float, age: int,

race: str, education: str,

location: str) -> List[IncomeComparison]:

"""Get multiple income comparisons for context"""

comparisons = []

# 1. National median comparison

national\_comparison = self.\_compare\_to\_national\_median(user\_income)

if national\_comparison:

comparisons.append(national\_comparison)

# 2. Racial demographic comparison

racial\_comparison = self.\_compare\_to\_racial\_demographic(user\_income, race)

if racial\_comparison:

comparisons.append(racial\_comparison)

# 3. Age group comparison

age\_comparison = self.\_compare\_to\_age\_group(user\_income, age, race)

if age\_comparison:

comparisons.append(age\_comparison)

# 4. Education level comparison

education\_comparison = self.\_compare\_to\_education\_level(user\_income, education, race)

if education\_comparison:

comparisons.append(education\_comparison)

# 5. Geographic comparison

location\_comparison = self.\_compare\_to\_location(user\_income, location, race)

if location\_comparison:

comparisons.append(location\_comparison)

return comparisons

def \_compare\_to\_national\_median(self, user\_income: float) -> Optional[IncomeComparison]:

"""Compare to overall U.S. median income"""

try:

# Try Census API first

median\_income = self.\_get\_census\_data('national\_median')

if not median\_income:

median\_income = self.fallback\_data['national\_median']

percentile = self.\_calculate\_percentile(user\_income, median\_income)

gap = median\_income - user\_income

gap\_percentage = (gap / median\_income) \* 100

if user\_income >= median\_income:

context = f"You earn {abs(gap\_percentage):.1f}% above the national median"

else:

context = f"You earn {gap\_percentage:.1f}% below the national median"

return IncomeComparison(

user\_income=user\_income,

comparison\_group="National Median (All Workers)",

median\_income=median\_income,

percentile\_rank=percentile,

income\_gap=gap,

gap\_percentage=gap\_percentage,

context\_message=context

)

except Exception as e:

print(f"Error in national comparison: {e}")

return None

def \_compare\_to\_racial\_demographic(self, user\_income: float, race: str) -> Optional[IncomeComparison]:

"""Compare to racial demographic median"""

try:

if race.lower() in ['african american', 'black']:

median\_income = self.\_get\_census\_data('african\_american\_median')

if not median\_income:

median\_income = self.fallback\_data['african\_american\_median']

group\_name = "African American Workers"

else:

# For other races, use national median as approximation

median\_income = self.fallback\_data['national\_median']

group\_name = f"{race} Workers"

percentile = self.\_calculate\_percentile(user\_income, median\_income)

gap = median\_income - user\_income

gap\_percentage = (gap / median\_income) \* 100

if user\_income >= median\_income:

context = f"You earn {abs(gap\_percentage):.1f}% above the median for {group\_name.lower()}"

else:

context = f"You earn {gap\_percentage:.1f}% below the median for {group\_name.lower()}"

return IncomeComparison(

user\_income=user\_income,

comparison\_group=group\_name,

median\_income=median\_income,

percentile\_rank=percentile,

income\_gap=gap,

gap\_percentage=gap\_percentage,

context\_message=context

)

except Exception as e:

print(f"Error in racial comparison: {e}")

return None

def \_compare\_to\_age\_group(self, user\_income: float, age: int, race: str) -> Optional[IncomeComparison]:

"""Compare to age group median"""

try:

# Determine age group

if 25 <= age <= 34:

age\_group = "25-34"

# Approximate median for African Americans 25-34

base\_median = 45000 if race.lower() in ['african american', 'black'] else 55000

elif 35 <= age <= 44:

age\_group = "35-44"

base\_median = 52000 if race.lower() in ['african american', 'black'] else 65000

else:

return None

percentile = self.\_calculate\_percentile(user\_income, base\_median)

gap = base\_median - user\_income

gap\_percentage = (gap / base\_median) \* 100

race\_qualifier = "African American " if race.lower() in ['african american', 'black'] else ""

if user\_income >= base\_median:

context = f"You earn {abs(gap\_percentage):.1f}% above the median for {race\_qualifier}workers ages {age\_group}"

else:

context = f"You earn {gap\_percentage:.1f}% below the median for {race\_qualifier}workers ages {age\_group}"

return IncomeComparison(

user\_income=user\_income,

comparison\_group=f"{race\_qualifier}Workers Ages {age\_group}",

median\_income=base\_median,

percentile\_rank=percentile,

income\_gap=gap,

gap\_percentage=gap\_percentage,

context\_message=context

)

except Exception as e:

print(f"Error in age comparison: {e}")

return None

def \_compare\_to\_education\_level(self, user\_income: float, education: str, race: str) -> Optional[IncomeComparison]:

"""Compare to education level median"""

try:

# Determine education group median

if 'bachelor' in education.lower() or 'college' in education.lower():

# College graduate median

base\_median = 65000 if race.lower() in ['african american', 'black'] else 78000

education\_group = "College Graduates"

elif 'master' in education.lower() or 'graduate' in education.lower():

base\_median = 75000 if race.lower() in ['african american', 'black'] else 90000

education\_group = "Graduate Degree Holders"

else:

return None

percentile = self.\_calculate\_percentile(user\_income, base\_median)

gap = base\_median - user\_income

gap\_percentage = (gap / base\_median) \* 100

race\_qualifier = "African American " if race.lower() in ['african american', 'black'] else ""

if user\_income >= base\_median:

context = f"You earn {abs(gap\_percentage):.1f}% above the median for {race\_qualifier}{education\_group.lower()}"

else:

context = f"You earn {gap\_percentage:.1f}% below the median for {race\_qualifier}{education\_group.lower()}"

return IncomeComparison(

user\_income=user\_income,

comparison\_group=f"{race\_qualifier}{education\_group}",

median\_income=base\_median,

percentile\_rank=percentile,

income\_gap=gap,

gap\_percentage=gap\_percentage,

context\_message=context

)

except Exception as e:

print(f"Error in education comparison: {e}")

return None

def \_compare\_to\_location(self, user\_income: float, location: str, race: str) -> Optional[IncomeComparison]:

"""Compare to local area median"""

try:

# Find metro area median

metro\_median = None

for metro, median in self.fallback\_data['metro\_medians'].items():

if metro.lower() in location.lower():

metro\_median = median

metro\_name = metro

break

if not metro\_median:

return None

# Adjust for racial demographic (approximately 0.68x for African Americans)

if race.lower() in ['african american', 'black']:

metro\_median = int(metro\_median \* 0.68)

percentile = self.\_calculate\_percentile(user\_income, metro\_median)

gap = metro\_median - user\_income

gap\_percentage = (gap / metro\_median) \* 100

race\_qualifier = "African American " if race.lower() in ['african american', 'black'] else ""

if user\_income >= metro\_median:

context = f"You earn {abs(gap\_percentage):.1f}% above the {metro\_name} median for {race\_qualifier}workers"

else:

context = f"You earn {gap\_percentage:.1f}% below the {metro\_name} median for {race\_qualifier}workers"

return IncomeComparison(

user\_income=user\_income,

comparison\_group=f"{metro\_name} {race\_qualifier}Workers",

median\_income=metro\_median,

percentile\_rank=percentile,

income\_gap=gap,

gap\_percentage=gap\_percentage,

context\_message=context

)

except Exception as e:

print(f"Error in location comparison: {e}")

return None

def \_get\_census\_data(self, data\_type: str) -> Optional[float]:

"""Fetch real-time data from Census API"""

try:

# This would implement actual Census API calls

# For now, return None to use fallback data

return None

except:

return None

def \_calculate\_percentile(self, user\_income: float, median\_income: float) -> int:

"""Estimate percentile rank based on income vs median"""

# Simplified percentile calculation

# In reality, you'd want more sophisticated distribution modeling

ratio = user\_income / median\_income

if ratio >= 2.0:

return 95

elif ratio >= 1.5:

return 85

elif ratio >= 1.2:

return 75

elif ratio >= 1.0:

return 60

elif ratio >= 0.8:

return 40

elif ratio >= 0.6:

return 25

else:

return 15

def generate\_motivational\_insights(self, comparisons: List[IncomeComparison],

target\_salary\_increase: float = 0.25) -> Dict:

"""Generate actionable insights and motivation"""

insights = {

'current\_position': [],

'opportunities': [],

'action\_items': []

}

# Analyze current position

for comp in comparisons:

if comp.percentile\_rank < 50:

insights['current\_position'].append(

f"You're in the {comp.percentile\_rank}th percentile for {comp.comparison\_group}"

)

# Calculate opportunity potential

current\_income = comparisons[0].user\_income if comparisons else 50000

target\_income = current\_income \* (1 + target\_salary\_increase)

insights['opportunities'].append(

f"A 25% salary increase would bring you to ${target\_income:,.0f}/year"

)

# Find the biggest income gap

if comparisons:

biggest\_gap = max(comparisons, key=lambda x: abs(x.income\_gap))

if biggest\_gap.income\_gap > 0:

insights['opportunities'].append(

f"Reaching the {biggest\_gap.comparison\_group} median would increase your income by ${biggest\_gap.income\_gap:,.0f}"

)

# Action items

insights['action\_items'] = [

"Use resume AI to find jobs with 15-30% salary increases",

"Focus on companies in higher-paying industries",

"Consider roles in higher-cost metros with remote work options",

"Invest in skills development for in-demand positions"

]

return insights

# Test the comparator

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

comparator = IncomeComparator()

# Test with sample user data

user\_income = 55000

comparisons = comparator.get\_comprehensive\_comparison(

user\_income=user\_income,

age=28,

race="African American",

education="Bachelor's Degree",

location="Atlanta"

)

print(f"Income Comparisons for ${user\_income:,}/year:")

print("="\*50)

for comp in comparisons:

print(f"\n{comp.comparison\_group}:")

print(f" Median: ${comp.median\_income:,}")

print(f" Your rank: {comp.percentile\_rank}th percentile")

print(f" {comp.context\_message}")

insights = comparator.generate\_motivational\_insights(comparisons)

print(f"\nInsights:")

for key, value in insights.items():

print(f"\n{key.title()}:")

for item in value:

print(f" • {item}")

**Step 2: Integration with Resume Analysis**

# Modified app.py additions

from income\_comparator import IncomeComparator, IncomeComparison

# Add to your existing Flask app

@app.route('/upload', methods=['POST'])

def upload\_resume():

# ... existing resume processing code ...

# NEW: Add income comparison

current\_salary = request.form.get('current\_salary', type=int)

age = request.form.get('age', type=int, default=30)

race = request.form.get('race', default='African American')

education\_level = request.form.get('education', default='Bachelor\'s Degree')

location = request.form.get('location', default='Atlanta')

income\_comparisons = []

motivational\_insights = {}

if current\_salary:

comparator = IncomeComparator()

income\_comparisons = comparator.get\_comprehensive\_comparison(

user\_income=current\_salary,

age=age,

race=race,

education=education\_level,

location=location

)

motivational\_insights = comparator.generate\_motivational\_insights(income\_comparisons)

return render\_template('results.html',

resume\_data=resume\_data,

job\_matches=job\_matches,

current\_salary=current\_salary,

income\_comparisons=income\_comparisons,

insights=motivational\_insights)

**Step 3: Enhanced User Input Form**

<!-- Enhanced upload.html template -->

<form method="POST" enctype="multipart/form-data">

<!-- Existing resume upload -->

<div class="form-group">

<label for="resume">Resume File (PDF or DOCX):</label>

<input type="file" name="resume" id="resume" accept=".pdf,.docx" required>

</div>

<!-- Enhanced demographic information -->

<div class="form-section">

<h3>Income Analysis (Optional but Recommended)</h3>

<p>Help us show you how your income compares and find the best opportunities</p>

<div class="form-row">

<div class="form-group">

<label for="current\_salary">Current Annual Salary:</label>

<input type="number" name="current\_salary" id="current\_salary"

placeholder="e.g., 55000" min="20000" max="200000">

</div>

<div class="form-group">

<label for="age">Age:</label>

<select name="age" id="age">

<option value="">Select Age</option>

<option value="25">25-27</option>

<option value="28">28-30</option>

<option value="31">31-33</option>

<option value="34">34-36</option>

<option value="37">37-40</option>

</select>

</div>

</div>

<div class="form-row">

<div class="form-group">

<label for="race">Race/Ethnicity:</label>

<select name="race" id="race">

<option value="African American">African American</option>

<option value="White">White</option>

<option value="Hispanic/Latino">Hispanic/Latino</option>

<option value="Asian">Asian</option>

<option value="Other">Other</option>

<option value="Prefer not to say">Prefer not to say</option>

</select>

</div>

<div class="form-group">

<label for="education">Education Level:</label>

<select name="education" id="education">

<option value="High School">High School</option>

<option value="Some College">Some College</option>

<option value="Bachelor's Degree">Bachelor's Degree</option>

<option value="Master's Degree">Master's Degree</option>

<option value="Doctoral Degree">Doctoral Degree</option>

</select>

</div>

</div>

</div>

<!-- Existing location dropdown -->

<div class="form-group">

<label for="location">Location:</label>

<select name="location" id="location">

<option value="Atlanta">Atlanta, GA</option>

<option value="Houston">Houston, TX</option>

<!-- ... other metro areas -->

</select>

</div>

<button type="submit" class="submit-btn">Analyze Resume & Compare Income</button>

</form>

**Step 4: Results Display Template**

<!-- Addition to results.html template -->

{% if income\_comparisons %}

<section class="income-analysis">

<h2>💰 Your Income Analysis</h2>

<div class="comparison-grid">

{% for comparison in income\_comparisons %}

<div class="comparison-card">

<h4>{{ comparison.comparison\_group }}</h4>

<div class="income-stats">

<div class="stat">

<span class="label">Your Income:</span>

<span class="value">${{ "{:,}".format(comparison.user\_income|int) }}</span>

</div>

<div class="stat">

<span class="label">Group Median:</span>

<span class="value">${{ "{:,}".format(comparison.median\_income|int) }}</span>

</div>

<div class="percentile-rank">

<span class="rank-number">{{ comparison.percentile\_rank }}th</span>

<span class="rank-label">percentile</span>

</div>

</div>

<div class="comparison-message {{ 'positive' if comparison.income\_gap < 0 else 'opportunity' }}">

{{ comparison.context\_message }}

</div>

{% if comparison.income\_gap > 1000 %}

<div class="opportunity-highlight">

<strong>💡 Opportunity:</strong> Reaching this median would increase your income by ${{ "{:,}".format(comparison.income\_gap|int) }}

</div>

**The goal is to produce the following:**

"Based on your resume, you're a 28-year-old African American with a Bachelor's degree earning $52,000 in Atlanta. You're in the 35th percentile for your demographic. The jobs we found could increase your income by 28%, moving you to the 70th percentile and adding $1,213/month to your cash flow."

**Resume Parser - Immediate Setup Guide**

**Step 1: Environment Setup (Next 30 minutes)**

**A. Create Project Directory**

Open your terminal/command prompt and run:

# Create and navigate to project directory

mkdir mingus-resume-ai

cd mingus-resume-ai

# Create Python virtual environment

python -m venv venv

# Activate virtual environment

# On Windows:

venv\Scripts\activate

# On Mac/Linux:

source venv/bin/activate

# Verify Python is working

python --version

**B. Install Required Packages**

# Install core packages one by one to avoid conflicts

pip install PyPDF2==3.0.1

pip install python-docx==0.8.11

pip install requests==2.31.0

pip install flask==2.3.3

pip install python-dotenv==1.0.0

# Verify installations

pip list

**C. Create Project Structure**

# Create directories

mkdir templates

mkdir static

mkdir uploads

mkdir test\_files

# Create main Python files

touch resume\_parser.py

touch app.py

touch config.py

touch .env

touch requirements.txt

# Verify structure

ls -la

**Step 2: Build the Resume Parser (Next 60 minutes)**

**A. Create resume\_parser.py**

Copy this exact code into your resume\_parser.py file:

import PyPDF2

import docx

import re

import os

from typing import Dict, List

class ResumeParser:

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

# Expanded skills list focused on your target market

self.skills\_keywords = [

# Technical skills

'python', 'javascript', 'java', 'sql', 'html', 'css', 'react',

'node.js', 'excel', 'powerpoint', 'word', 'outlook', 'salesforce',

# Business skills

'project management', 'data analysis', 'marketing', 'sales',

'customer service', 'accounting', 'finance', 'budgeting',

# Soft skills

'communication', 'leadership', 'teamwork', 'problem solving',

'time management', 'organization', 'presentation', 'training'

]

def extract\_text\_from\_file(self, file\_path: str) -> str:

"""Extract text from PDF or DOCX files"""

print(f"Processing file: {file\_path}")

try:

if file\_path.lower().endswith('.pdf'):

return self.\_extract\_from\_pdf(file\_path)

elif file\_path.lower().endswith('.docx'):

return self.\_extract\_from\_docx(file\_path)

else:

raise ValueError("Unsupported file format. Use PDF or DOCX.")

except Exception as e:

print(f"Error extracting text: {e}")

return ""

def \_extract\_from\_pdf(self, file\_path: str) -> str:

"""Extract text from PDF"""

text = ""

try:

with open(file\_path, 'rb') as file:

pdf\_reader = PyPDF2.PdfReader(file)

print(f"PDF has {len(pdf\_reader.pages)} pages")

for page\_num, page in enumerate(pdf\_reader.pages):

page\_text = page.extract\_text()

text += page\_text + "\n"

print(f"Extracted {len(page\_text)} characters from page {page\_num + 1}")

except Exception as e:

print(f"Error reading PDF: {e}")

return text

def \_extract\_from\_docx(self, file\_path: str) -> str:

"""Extract text from DOCX"""

text = ""

try:

doc = docx.Document(file\_path)

print(f"DOCX has {len(doc.paragraphs)} paragraphs")

for paragraph in doc.paragraphs:

text += paragraph.text + "\n"

except Exception as e:

print(f"Error reading DOCX: {e}")

return text

def parse\_resume(self, file\_path: str) -> Dict:

"""Main parsing function - returns structured data"""

print(f"Starting to parse resume: {file\_path}")

text = self.extract\_text\_from\_file(file\_path)

if not text.strip():

print("Warning: No text extracted from file")

return {

'error': 'Could not extract text from file',

'name': 'Unknown',

'email': 'Not found',

'phone': 'Not found',

'experience': [],

'education': [],

'skills': [],

'raw\_text': ''

}

print(f"Extracted {len(text)} characters total")

parsed\_data = {

'name': self.\_extract\_name(text),

'email': self.\_extract\_email(text),

'phone': self.\_extract\_phone(text),

'experience': self.\_extract\_experience(text),

'education': self.\_extract\_education(text),

'skills': self.\_extract\_skills(text),

'raw\_text': text[:1000] + "..." if len(text) > 1000 else text # Truncate for display

}

print(f"Parsing complete. Found {len(parsed\_data['skills'])} skills")

return parsed\_data

def \_extract\_name(self, text: str) -> str:

"""Extract name (first non-empty line usually)"""

lines = [line.strip() for line in text.split('\n') if line.strip()]

for line in lines[:5]: # Check first 5 lines

# Skip lines that look like contact info

if not any(keyword in line.lower() for keyword in

['email', '@', 'phone', 'address', 'linkedin', 'github', 'http']):

# Skip lines that are too long (probably not a name)

if len(line) < 50 and len(line.split()) <= 4:

return line

return "Name not found"

def \_extract\_email(self, text: str) -> str:

"""Extract email address"""

email\_pattern = r'\b[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'

emails = re.findall(email\_pattern, text)

return emails[0] if emails else "Email not found"

def \_extract\_phone(self, text: str) -> str:

"""Extract phone number"""

# Multiple phone patterns

patterns = [

r'(\+?1[-.\s]?)?\(?([0-9]{3})\)?[-.\s]?([0-9]{3})[-.\s]?([0-9]{4})',

r'(\d{3})[-.](\d{3})[-.](\d{4})',

r'\((\d{3})\)\s\*(\d{3})[-.](\d{4})'

]

for pattern in patterns:

phones = re.findall(pattern, text)

if phones:

# Format the phone number

if isinstance(phones[0], tuple):

return ''.join(phones[0])

return phones[0]

return "Phone not found"

def \_extract\_experience(self, text: str) -> List[str]:

"""Extract work experience"""

experience\_keywords = [

'experience', 'employment', 'work history', 'professional experience',

'career history', 'work experience'

]

lines = text.split('\n')

experience\_section = []

in\_experience = False

for i, line in enumerate(lines):

line = line.strip()

# Check if we're entering experience section

if any(keyword in line.lower() for keyword in experience\_keywords):

in\_experience = True

continue

# Check if we're leaving experience section

elif in\_experience and any(keyword in line.lower() for keyword in

['education', 'skills', 'certifications', 'projects']):

break

# If we're in experience section and line looks like a job

elif in\_experience and line:

# Look for job titles (lines with years, company names, or job-like words)

if (re.search(r'\d{4}', line) or # Contains year

any(word in line.lower() for word in

['inc', 'llc', 'corp', 'company', 'university', 'hospital']) or

any(title in line.lower() for title in

['manager', 'analyst', 'specialist', 'coordinator', 'assistant',

'developer', 'engineer', 'consultant', 'representative'])):

experience\_section.append(line)

return experience\_section[:10] # Return up to 10 most relevant

def \_extract\_education(self, text: str) -> List[str]:

"""Extract education information"""

education\_keywords = [

'education', 'degree', 'university', 'college', 'bachelor',

'master', 'phd', 'doctorate', 'associates'

]

lines = text.split('\n')

education\_section = []

in\_education = False

for line in lines:

line = line.strip()

# Check if we're entering education section

if any(keyword in line.lower() for keyword in education\_keywords):

in\_education = True

# If this line itself contains degree info, add it

if any(degree in line.lower() for degree in

['bachelor', 'master', 'phd', 'degree', 'associates']):

education\_section.append(line)

continue

# If we're in education section

elif in\_education and line:

# Add lines that look like education

if (any(word in line.lower() for word in

['university', 'college', 'institute', 'school']) or

any(degree in line.lower() for degree in

['bachelor', 'master', 'phd', 'bs', 'ba', 'ms', 'ma']) or

re.search(r'\d{4}', line)): # Contains graduation year

education\_section.append(line)

# Stop if we hit another section

elif any(keyword in line.lower() for keyword in

['experience', 'skills', 'certifications']):

break

return education\_section[:5] # Return up to 5 entries

def \_extract\_skills(self, text: str) -> List[str]:

"""Extract skills from resume"""

found\_skills = []

text\_lower = text.lower()

print("Searching for skills...")

for skill in self.skills\_keywords:

if skill in text\_lower:

found\_skills.append(skill.title())

print(f"Found skill: {skill}")

# Also look for programming languages and tools

additional\_skills = [

'c++', 'c#', 'php', 'ruby', 'swift', 'kotlin', 'typescript',

'mysql', 'postgresql', 'mongodb', 'aws', 'azure', 'docker',

'git', 'jira', 'slack', 'zoom', 'teams'

]

for skill in additional\_skills:

if skill in text\_lower:

found\_skills.append(skill.upper() if skill in ['aws', 'sql'] else skill.title())

# Remove duplicates while preserving order

unique\_skills = []

for skill in found\_skills:

if skill not in unique\_skills:

unique\_skills.append(skill)

return unique\_skills

# Test function

def test\_parser():

"""Test the parser with a sample file"""

parser = ResumeParser()

# Test with your own resume file

test\_file = input("Enter path to your test resume file (PDF or DOCX): ")

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

result = parser.parse\_resume(test\_file)

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

print("RESUME PARSING RESULTS")

print("="\*50)

for key, value in result.items():

if key != 'raw\_text': # Skip raw text for cleaner output

print(f"\n{key.upper()}:")

if isinstance(value, list):

for item in value:

print(f" - {item}")

else:

print(f" {value}")

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

else:

print(f"File not found: {test\_file}")

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

test\_parser()

**B. Test the Resume Parser Right Now**

1. **Save a test resume** to your test\_files folder (use your own resume or download a sample PDF/DOCX)
2. **Run the parser**:

python resume\_parser.py

1. **When prompted, enter the path to your test file**:

# Example paths:

test\_files/sample\_resume.pdf

test\_files/my\_resume.docx

**C. Expected Output**

You should see something like:

Processing file: test\_files/sample\_resume.pdf

PDF has 1 pages

Extracted 1247 characters from page 1

Starting to parse resume: test\_files/sample\_resume.pdf

Extracted 1247 characters total

Searching for skills...

Found skill: python

Found skill: excel

Found skill: project management

Parsing complete. Found 3 skills

==================================================

RESUME PARSING RESULTS

==================================================

NAME:

John Smith

EMAIL:

john.smith@email.com

PHONE:

(555) 123-4567

EXPERIENCE:

- Software Developer at ABC Company 2020-2023

- Junior Analyst at XYZ Corp 2018-2020

EDUCATION:

- Bachelor of Science in Computer Science

- University of Technology 2018

SKILLS:

- Python

- Excel

- Project Management

==================================================

**Step 3: Quick Fixes if Something Goes Wrong**

**A. Common Issues and Solutions**

**Issue: "ModuleNotFoundError: No module named 'PyPDF2'"**

# Solution: Reinstall packages

pip uninstall PyPDF2

pip install PyPDF2==3.0.1

**Issue: "Permission denied" or file access errors**

# Solution: Check file permissions

ls -la test\_files/

# Make sure your resume file is readable

chmod 644 test\_files/your\_resume.pdf

**Issue: "No text extracted from file"**

* Try a different PDF (some PDFs are image-based)
* Test with a Word document instead
* Use a simple text-based resume

**Issue: Python virtual environment not working**

# Alternative setup without virtual environment

pip install --user PyPDF2 python-docx requests flask python-dotenv

**B. Debugging Steps**

If the parser isn't finding skills/info correctly:

1. **Check raw text extraction**:

# Add this to test\_parser() function:

print(f"\nRAW TEXT (first 500 chars):")

print(result['raw\_text'][:500])

1. **Test with a simple resume**: Create a simple Word document with:

* Your name at the top
* Email and phone
* "Work Experience" section
* "Skills: Python, Excel, Project Management"

**Step 4: Build Basic Web Interface (Next 30 minutes)**

**A. Create Simple Flask App (app.py)**

from flask import Flask, render\_template, request, redirect, url\_for, flash

import os

from werkzeug.utils import secure\_filename

from resume\_parser import ResumeParser

app = Flask(\_\_name\_\_)

app.secret\_key = 'mingus-dev-key' # Change in production

app.config['UPLOAD\_FOLDER'] = 'uploads'

app.config['MAX\_CONTENT\_LENGTH'] = 5 \* 1024 \* 1024 # 5MB

ALLOWED\_EXTENSIONS = {'pdf', 'docx'}

parser = ResumeParser()

def allowed\_file(filename):

return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED\_EXTENSIONS

@app.route('/')

def index():

return '''

<h1>Mingus Resume Parser - MVP</h1>

<form action="/upload" method="post" enctype="multipart/form-data">

<p>

<label>Upload Resume (PDF or DOCX):</label><br>

<input type="file" name="resume" accept=".pdf,.docx" required>

</p>

<p>

<input type="submit" value="Analyze Resume">

</p>

</form>

'''

@app.route('/upload', methods=['POST'])

def upload\_resume():

if 'resume' not in request.files:

return 'No file uploaded', 400

file = request.files['resume']

if file.filename == '':

return 'No file selected', 400

if file and allowed\_file(file.filename):

filename = secure\_filename(file.filename)

filepath = os.path.join(app.config['UPLOAD\_FOLDER'], filename)

file.save(filepath)

try:

# Parse the resume

result = parser.parse\_resume(filepath)

# Clean up file

os.remove(filepath)

# Display results

html = f'''

<h1>Resume Analysis Results</h1>

<h2>Basic Info</h2>

<p><strong>Name:</strong> {result['name']}</p>

<p><strong>Email:</strong> {result['email']}</p>

<p><strong>Phone:</strong> {result['phone']}</p>

<h2>Skills Found ({len(result['skills'])})</h2>

<ul>

'''

for skill in result['skills']:

html += f'<li>{skill}</li>'

html += f'''

</ul>

<h2>Experience ({len(result['experience'])} entries)</h2>

<ul>

'''

for exp in result['experience']:

html += f'<li>{exp}</li>'

html += '''

</ul>

<p><a href="/">← Analyze Another Resume</a></p>

'''

return html

except Exception as e:

if os.path.exists(filepath):

os.remove(filepath)

return f'Error processing resume: {str(e)}', 500

return 'Invalid file type', 400

if \_\_name\_\_ == '\_\_main\_\_':

# Create uploads directory

os.makedirs('uploads', exist\_ok=True)

app.run(debug=True, port=5000)

**B. Test Web Interface**

# Run the web app

python app.py

# Open browser to http://localhost:5000

# Upload a resume and see results

**Step 5: Validate It Works (Next 15 minutes)**

**A. Success Checklist**

✅ Resume parser extracts text from PDF/DOCX  
✅ Parser finds name, email, phone  
✅ Parser identifies skills from resume  
✅ Parser finds work experience entries  
✅ Web interface accepts file uploads  
✅ Web interface displays parsed results

**B. Quick Validation Test**

1. Upload 2-3 different resume formats
2. Check that skills are being found correctly
3. Verify experience and education sections work
4. Make sure web interface doesn't crash

**Next Steps (Tomorrow)**

Once this is working, you'll add:

1. **Job search integration** (Adzuna API - free)
2. **Better web interface** (HTML templates)
3. **Job matching algorithm** (skills comparison)

**Total time invested so far: ~2.5 hours**  
**Total cost so far: $0**  
**Working features: Complete resume analysis**

This gives you a solid foundation to build on and proves the concept works before investing more time or money!