

Final Project Report: Automated Resume Screening & Ranking

Arsh Syed

Naveen Jindal School of Management, University of Texas at Dallas

BUAN 6385.S01 - Robotic Process Automation

Dr. Talukdar S Asgar

May 10, 2025

1. Introduction

I was first fascinated by the potential of Robotic Process Automation (RPA) when I came across it. Fundamentally, RPA offers a thrilling and pragmatic promise: the ability to build digital systems that mimic human actions to carry out tasks quickly, accurately, and consistently. It's not science fiction; it's intelligent automation and it's now changing whole sectors. Simultaneously fascinating and taxing was the idea that I could develop a bot to simplify a complex process—especially one that usually requires human perspective and decision-making. The basis for this initiative was my interest and drive.

In today's digital world, automation has transitioned from being a luxury to a necessity. Companies are handling ever-increasing volumes of data, therefore the demand for smarter and faster processes has grown significantly. Manual operations slow down teams, create discrepancies, and ultimately limit scalability. RPA fills that need by allowing people to focus on more strategic decision-making by assigning rule-based, repetitive duties to software robots.

In choosing a focal point for my final project, I deliberately avoided overly easy situations like sending emails or automating file transfers. Instead, I wanted to explore a real-world bottleneck — something both complex and widely relevant. That's when I started thinking about recruitment.

Resume screening is widely recognized as one of the most time-consuming stages of the hiring process, a fact I have observed firsthand. Recruiters receive hundreds, sometimes thousands, of applications for a single position. Even with applicant tracking systems (ATS), it's nearly impossible to manually review every resume in depth. Often, resumes get skimmed, and great candidates may be passed over simply because their formatting or keywords didn't align with what the recruiter was expecting.

The issue of inconsistency. Two recruiters might look at the same CV and arrive at rather different insights. Unconscious and systematic bias may intrude in. Although not every human mistake can be fixed by automation, I came to see that it could help the first screening process become organized, fair, and efficient.

That started the basis for my project:

Can I construct a robot that scans and assesses resumes according to a job description—and returns a ranked, scored list of candidates based on relevance? This concerns enhancing human judgment, not replacing it. Imagine it as the first filter: the bot does the hard work of gathering important information, matching it against preset criteria, and organizing the findings so that recruiters can readily go through it. In this approach, recruiters only spend time reviewing the most qualified applicants.

Resume Screening: Why?

Among all potential automated processes, resume screening highlighted a few reasons:

- It is quite repetitive: Examining resumes entails the same processes repeatedly extracting information, matching it against a JD, and grading relevancy.
- Resumes might be Word or PDF files, may have varied formats, but ultimately include similar categories of information in semi-structured form.
- When performed manually, it is prone to mistakes: Human fatigue causes uneven screening, overlooked details, and selection biases.
- Once automated, this procedure is scalable and can manage 10 or 10,000 resumes with little modification.

The choice was obvious and using UiPath—among the most potent RPA tools currently available—I was eager to bring this concept to life.

Framing the Bot

From the very beginning I realized this bot couldn't be a one-size-fits-all solution. The practical use calls for flexibility. Positions evolve. Different roles call different sets of talents. The position an individual holds determines the relative importance of experience and education. Therefore, the system I envisioned had to be responsive, adaptable, and modular.

This is what I hoped to achieve:

- Read each resume file (PDF format) and extract important fields (name, email, experience, degree, skills).
- Load a reference file with the job description and list of needed qualifications for a particular position.
- Match the reference job file's resume data with
- Give the candidate a hybrid model score combining experience, education, and skills.
- Sort the results by best match in a neat Excel file.
- Guarantee reusability: By just altering the reference file—no re-coding necessary—the bot should function for any job category.

The Obstacles I Expected

I already expected some very big obstacles before even starting to write any logic. First, resumes are very erratic. Two resumes might appear utterly different even for related positions. While some candidates vaguely characterize their experience, others enumerate

it in years. Some simply say "graduate education"; others clearly write "M.Sc. in Data Science." Extracting useful data needed more than just OCR; it required pattern recognition, keyword matching, and in some instances, informed guesses.

Designing the scoring system would second require great attention. For example, if a candidate possesses three of the five skills required for the position, is that a match of 60%? What if they possess uncommon talents not mentioned but still applicable? I came to see that I needed a scoring system that was understandable and adaptable—one that balanced various criteria.

Third, I had to consider ideally how to construct this using top standards. We studied in class several process modeling methodologies - flowcharts, sequences, and state machines. I hoped my final project to not just work properly but also exhibit appropriate RPA design. I decided on a State Machine model for this reason: it made transitions between data extraction, comparison, scoring, and output much more graceful.

Looking Forward When implemented deliberately, it became a genuine investigation of what automation could accomplish. I came to see that RPA is about creating better workflows, decreasing human mistake, and producing consistent results rather than just removing manual work. And I had complete control from planning, data curation, logic building, exception handling, testing, and optimization.

I will cover each step in meticulous detail in the following chapters. From the way I set my input files to the exact automations I utilized in UiPath, how I tested the workflow, managed edge cases, and constantly enhanced the scoring algorithm. To provide a whole picture, I have also included graphics such flowcharts, test screenshots, and final output Excel files.

By the end of this report, I hope anyone—even someone without a thorough technical background—will be able to see how an RPA bot may intelligently manage one of the most boring yet crucial chores in business today and maybe even be motivated to create one of their own.

2. Problem Statement

When I started to determine what I wanted to automate for my RPA final project, I knew I didn't want to accept a standard application case. There were many of ways to show my knowledge of UiPath—invoice processing, web scraping, email automation—all legitimate but not particularly inspiring for me personally. I wished to concentrate on a situation that really shows an industry inefficiency—something I have either experienced or seen closely. The answer became obvious after some consideration: resume screening.

One of those areas where everyone believes they have a process—until you examine closely and discover it full of bottlenecks, subjective judgments, and time-consuming manual work—is recruitment. Even in businesses using Applicant Tracking Systems (ATS), the resume review phase is still mostly human-driven. The reality is that although ATS systems lack nuance, they can filter based keywords. They surely don't rank or score candidates in a way that feels complete; they lack understanding of context.

Let's explore a common situation: a recruiter receives 400–600 resumes after posting a position for a Senior Data Analyst or Social Media Manager. They have to open every file now, scan for the appropriate experience, verify education, pinpoint critical skills, and psychologically judge whether the resume “feels” like a good match. Certain resumes are badly written. Others bury vital knowledge. Some simply expect you to deduce years of experience from job descriptions; others do not even mention it at all.

This is where contradictions start. Should one recruiter go through 200 resumes at 9 a.m., they may be awake and thorough. After a full day of meetings, they might skim or skip if they are looking at another 200 at five p.m. That variation causes inefficiencies, bias, and missed applicants. Going into this project, my basic hypothesis was that a properly built

RPA bot would be able to manage the initial screening phase more reliably than a person would.

I'm not implying here that automation should make ultimate hiring decisions — far from it. The process will always include human judgment, team fit, and interpersonal skills. However, that is a domain where automation excels when it comes to technically, data-driven comparing resumes: matching degrees, tallying years of experience, checking necessary credentials.

Early on I came to see that resume screening involves far more than merely file reading. It's about interpreting semi-structured data, knowing context, and evaluating relevance versus a moving target (the job description). Should I be able to break that—even for a small dataset—I would provide significant worth.

Why Manual Resume Screening Fails at Scale

Let me break down some of the specific pain points I wanted my bot to solve:

- **Volume Overload:** Human recruiters can only process a limited number of resumes per hour. That means many great candidates are missed simply because their resume is somewhere in the bottom half of the stack.
- **Format Inconsistency:** Resumes come in wildly different formats. Some use headers like “Professional Summary,” others say, “Work History.” Some have degrees on page one; others bury it on page three. Parsing these manually is time-consuming.
- **Keyword Dependency:** Most ATS tools rely on keyword matching, but not every candidate uses the same terms. One might say “Machine Learning,” another “ML.” A recruiter might miss that they’re the same unless they know what to look for.

- **Unconscious Bias:** Human screeners may be swayed by resume design, name, university, or even location. Automation doesn't make assumptions — it evaluates based on what it's programmed to detect.
- **Inconsistent Scoring:** Let's say two recruiters are reviewing the same resume. One might say, "This person has five years of experience, that's great." The other might say, "But they don't have Tableau — pass." A bot can apply a consistent, predefined formula to score every candidate equally.

My project aimed to replace this unstructured first pass with a structured, rule-based evaluation system — a system that works tirelessly, fairly, and transparently.

What Makes This Problem Complex

One of the misconceptions I encountered while discussing my project idea with peers was the assumption that resume parsing is easy — "Just read the file and look for keywords." If only it were that simple.

Here's what makes this problem harder than it looks:

- **Resumes don't follow a standard structure:** Unlike spreadsheets or forms, resumes are designed for humans, not machines. Parsing them reliably means anticipating dozens of ways people might describe the same thing.
- **Skills are fuzzy:** It's not enough to search for the word "Python." A resume might say "Proficient in scripting languages including Python, R, and Bash" or "Built data pipelines using Airflow and Pandas." The skill is there — but it's wrapped in a sentence.
- **Experience isn't always quantified:** Some candidates write "Worked at XYZ Corp since 2017." The number of years isn't explicitly written. My bot needs to detect patterns, use regular expressions, and sometimes make educated assumptions.

- Degrees come in multiple forms: A person with a “MSc in Information Systems” is equivalent to someone with a “Master of Science in IS.” The bot must normalize such variations before scoring them.
- Scoring logic must be fair and explainable: If someone asks why Candidate A scored higher than Candidate B, the bot’s formula must be transparent. This means I needed a hybrid model that considers skills match %, degree weighting, and years of experience — not just a binary pass/ fail.

Not in the AI sense, but in how it managed variability in human-written documents, all of this meant I had to create an RPA bot that was resilient, flexible, and smart.

It has very great relevance to the sector as well. This issue goes beyond the classroom. Companies spend billions trying to simplify the talent acquisition pipeline in the real world, especially in HR and recruiting technology. From LinkedIn’s AI-based solutions to niche ATS startups, everyone is working to fix this issue. Most tools, though, are costly black-box models with minimal openness.

Built in UiPath, my bot is open, inspectable, and alterable. It depends on transparent, rule-based logic anyone can adjust according to evolving job needs rather than on machine learning.

This also makes it ideal for small and medium-sized businesses—those that lack the resources for sophisticated AI solutions yet still want to provide speed and consistency to hiring. Still, I had to put some boundaries here as I wanted a practical bot and not a pioneered bot that can take it all like ATS.

- Process a batch of resumes from a folder.
- Extract key data fields (name, title, email, degree, experience, skills).
- Compare them to a provided job description (in a reference .txt file).

- Score and rank them based on a weighted formula.
- Export the results into an Excel file with relevant columns.

Due the scope being less I could concentrate on performance, reusability, and correctness.

As I wanted to ensure the automation operated perfectly.

3. Data Use & Input Process

I first assumed that the simple part of this project would be data gathering and processing. Resumes are after all papers we all have seen before; job descriptions are generally little more than a brief list of responsibilities and criteria. But I swiftly discovered how deceptive this assumption could be—because in the field of automation, the presentation of data matters as much as its content.

I will go over in detail in this section the kinds of input I dealt with, the techniques I employed to organize this data, the particular design choices I took while readying it for automation, and why handling such "human-generated" data was significantly more difficult than I first thought.

3.1 Resume Data: Unstructured and Inconsistent

The core dataset for this project consisted of resume PDFs from real-world inspired profiles. These documents were not formatted the same way, nor were they built with machine-readability in mind. This inconsistency made the task both realistic and challenging, which was exactly what I wanted. After all, real-world automation needs to handle real-world messiness.

- Some resumes were clean and followed a logical sequence — starting with the candidate's name, followed by contact information, summary, experience, education, and skills. But others broke all expectations:

- Some resumes opened with a job title, not a name.
- In some, contact information was embedded in footers or written vertically along the side in a design-heavy layout.
- Dates of employment weren't always standard. I saw "March 2019 to Present," "2019 – Now," and even just "3+ years."
- Skills were written as full sentences, not lists — requiring contextual extraction.
- Others listed "Technologies Used" inside dense paragraphs under experience — with no consistent delimiter.

These inconsistencies made it impossible to rely on fixed rules. I had to build smart, adaptive parsing logic that allowed flexibility in interpreting the text.

Figure 3.1: A side-by-side image of two parsed resumes (one clean and list-based; another embedded in paragraph form).

<p>Richard Howard Head of Supply Chain</p> <p>Professional Summary: Strategic Supply Chain Head with 15+ years leading global procurement, logistics, and S&OP functions. Expert in cost reduction, vendor negotiations, and ERP transformations.</p> <p>Skills: S&OP, Global Sourcing, ERP (SAP/Oracle), Vendor Negotiation, Logistics Optimization, Procurement Strategy, Forecasting, Cost Control</p> <p>Experience: Head of Supply Chain - Allied Logistics (2016 - Present) <ul style="list-style-type: none"> - Reduced procurement costs by 18% over 3 years through strategic sourcing. - Led global supply chain restructuring across 4 continents. - Oversaw ERP upgrade to SAP S/4HANA for logistics and inventory workflows. Supply Chain Manager - GlobalSource Ltd. (2010 - 2016) <ul style="list-style-type: none"> - Managed \$80M spend portfolio across raw materials and finished goods. - Established key supplier SLAs and improved OTIF by 20%. Education: MBA in Supply Chain Management, Kellogg School of Management, 2009</p>	<p>CONSULTANT Profile</p> <p>A highly accomplished, skilled and talented Consulting manager with a thorough understanding of the Software development Life Cycle and a proven track record of successfully providing overall direction to project teams and managing client relationships.</p> <p>Professional Experience Consultant , 01/2001 to 11/2014 Company Name 1/4 City , State</p> <ul style="list-style-type: none"> • Led a data migration project for a client P & C insurance company from a legacy client server insurance system with a relational backend to a n-tiered insurance system which involved the entire gamut of activities from defining requirements, selecting the application tool suite to use for the data migration, performing data mapping and gap analysis, designing the technical approach, performing the necessary business analysis, engaging with the target system vendor technical leads to ensure structural compatibility, developing the transformation, testing the transformation all the way to the upload into the target system, supporting the UAT and creating a deployment plan Led and managed the design, development and implementation of a Billing module of a P & C system to support a change in the lockbox financial institution Served as a Technical Project Manager as well as a vendor relationship manager on a Commercial Lines Expansion Project, a key business initiative of the client company to enable the organization to underwrite business in additional states. • This CL Expansion effort is expected to capture 1.5% of the commercial lines market in each additional state (about \$35,000,000) by the end of 2011 Led and managed a cross functional team to successfully migrate existing processes from an in-house personal lines system to a system in a hosted environment Led client efforts to support external IS audit conducted by Ernst & Young Led and managed the design, development and deployment of a print solution using Accelo suite of products (now part of the Adobe suite of products) to replace an existing print sub-system for a P & C Insurance system. • Conducted an internal practice-wide training to prepare other teams to undertake such print solution development efforts for other P & C clients Led and managed the design, development and integration of a custom built compliance software sub-system to help a P & C Insurance system comply with the requirements of the Office of Foreign Assets Control (OFAC) Assisted the IT Director at a client company in migrating their existing legacy system to the POINT IN system. • This was a full service engagement that included, analyzing and determining the hardware requirements for the selected system, negotiating the hardware purchase (saved the client over 40% on the original hardware purchase price), overseeing the hardware installation, developing the UAT plan, overseeing the UAT, advising and assisting the client IT Director on the project. • Conducted an analysis of the popular CRM software packages and presented the findings to the client's senior management team as part of an effort to facilitate their selection of a CRM package Technically led the design and development of an Imaging prototype system Provided technical leadership in re-designing the Gay Carpenter report to ensure accuracy and better data processing Provided technical leadership for migrating a Commercial lines system to a new payment vendor Provided production support on several client engagements Mentored junior level staff on several system development efforts Provided expertise and oversight in the development of marketing collateral Participated in a technical advisory role in determining the feasibility and the subsequent techno-business analysis of implementing an e-bill presentment module for a personal lines system which involved a lockbox vendor replacement. <p>Software Engineer , 01/2000 to 01/2001 Company Name 1/4 City , State</p> <ul style="list-style-type: none"> • Designed, developed and tested insurance applications. • Maintained existing insurance applications. <p>Sr. Applications Analyst Programmer , 11/1997 to 12/2000 Company Name 1/4 City , State</p> <ul style="list-style-type: none"> • Designed, developed and tested applications using Centura/SQL Windows as the frontend and Oracle as the backend. • Translated business requirements into technical specifications. • Formed a part of the core group on several teams for customizing and deploying the company's base P & C Insurance system Implemented Aviation, Tourism and Garage Liability lines of business for Sud America using Centura as the front end and Oracle as the back end • Implemented Lead and Adoptions as well as Engineering lines of business for Zurich American using SQL Windows as front end and Oracle as the back end Formed a part of the core group of a successful team that implemented an insurance auto-warranty system for General Motors Corporation using Centura Developer for the front-end and Oracle as the back-end Participated in responding to RFPs as an SME Led the conversion of a 16-bit WPC (P&C Insurance) system as a technical advisor to its 32-bit version Mentored junior level staff on application development efforts. <p>Education Master's degree : Computer Science , 2000 University of South Carolina 1/4 City , State , USA Computer Science Bachelor's degree : Physics , 1992 Gujarat University 1/4 City , State , India Physics Accomplishments <ul style="list-style-type: none"> • Cross functional leadership and management Served as a Technical Project Manager as well as a vendor relationship manager on a Commercial Lines Expansion Project, a key business initiative of the client company that was aimed at capturing 1.5% of the commercial lines market in each state (about \$35,000,000) by the end of 2011 Process migration leadership Led and managed a cross functional team </p>
--	---

3.2 Job Description Data: Reference Files in .txt Format

On the other side of the automation were the job descriptions. These were manually curated plain text files, each representing a single role that a recruiter or HR team would typically post. Each job file followed a consistent format I defined:

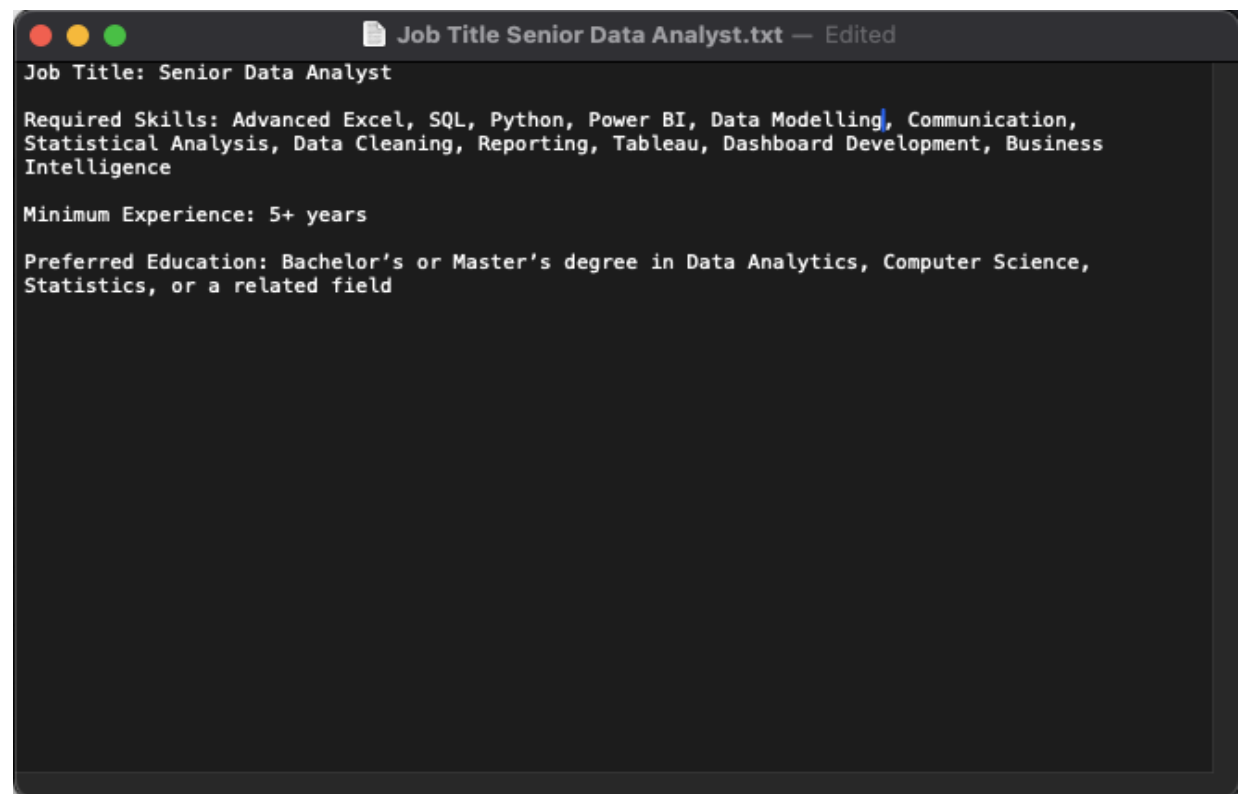
- Job Title (used for semantic comparison)
- Required Experience
- List of Key Skills
- Preferred Certifications or Keywords

Since this component of the input data was the reference norm for scoring candidates, I wanted to make sure it was organized. Every automation logic stemmed from how closely each resume matched this job file.

Every time I ran the bot, I might just switch the reference file — from Senior Data Analyst to SEO Specialist, for instance — and the bot would read the new set of skills, title, and experience requirements, then rerun the scoring logic.

This design choice let me simulate a hiring pipeline for many job roles, making my automation modular, testable, and scalable.

Figure 3.2: Reference File



3.3 Parsing the Resume Text

Most of the input processing logic relied on converting PDF resumes to raw text.

UiPath's Read PDF Text activity became the primary tool for this. I tested both OCR-based extraction (Read PDF with OCR) and native text parsing. OCR was necessary for scanned or image-based resumes, but I soon discovered that most resumes used in this project were digital PDFs — so native extraction was faster, cleaner, and more accurate.

One challenge I faced was determining how to split the resume into manageable chunks.

Eventually, I settled on:

```
lines = resumeText.Split(Environment.NewLine.ToCharArray,  
StringSplitOptions.RemoveEmptyEntries)
```

This gave me a list of non-empty lines, which I could then scan line-by-line for key attributes like name, email, title, education, and skills. But even this line-splitting approach had to be handled with caution. Some resumes had inconsistent spacing or embedded formatting (e.g., line breaks between job title and company name). I learned to not hardcode assumptions like “name is always on line 1” or “skills are listed after the word ‘Skills.’”

I added fallback logic — if name detection failed in the top 3 lines, for instance, I looked deeper or even fell back to using the filename (as a last resort). This adaptability became one of the core strengths of my approach.

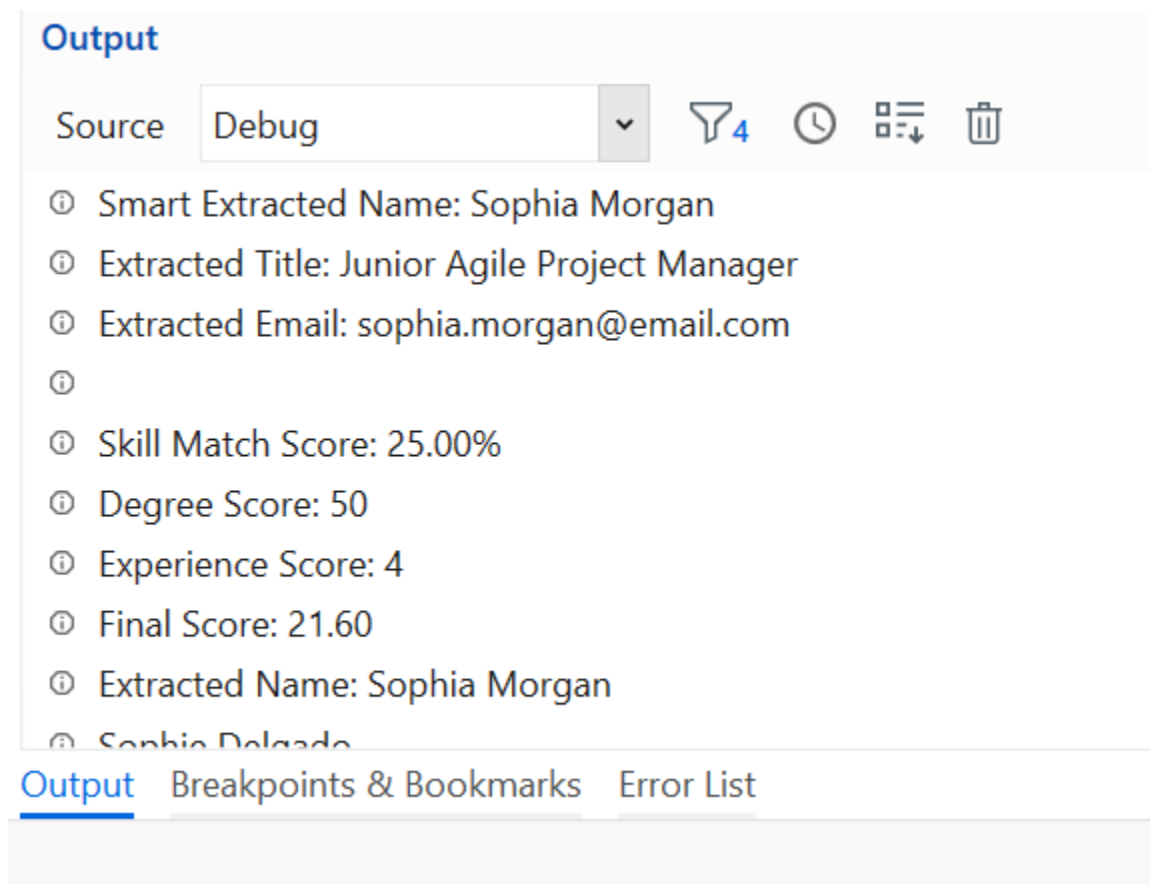
3.4 Extracting Relevant Fields from Resumes

Every resume, no matter how chaotic, had five key pieces of information I needed to extract:

Field	Extraction Method
Name	From top lines, checked against job title keywords to avoid false matches
Email	Regex: <code>[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}</code>
Job Title	Matched against predefined title keywords (e.g., “Analyst,” “Manager,”
Skills	Matched line-by-line against job reference skill list
Experience	Regex for “X years” or parsed from employment date ranges

This logic wasn’t just written once — it was continuously improved based on what I observed while debugging the bot.

Figure 3.4: Annotated image of parsed lines from a resume showing name, email, and detected skills.



3.5 Organizing Resume and Reference Files

To keep the automation simple to run, I structured all inputs as follows:

- ResumesFolder → contains PDF files like Sophia_Adams_DevOps.pdf
- ReferenceFiles → contains .txt files like Job Title social media Manager.txt
- Output → the Excel file CandidateScores.xlsx was created here after each run

The `Directory.GetFiles()` activity allowed my bot to dynamically fetch every resume. This meant I didn't have to manually input paths — the bot could scale to hundreds of resumes in a batch process.

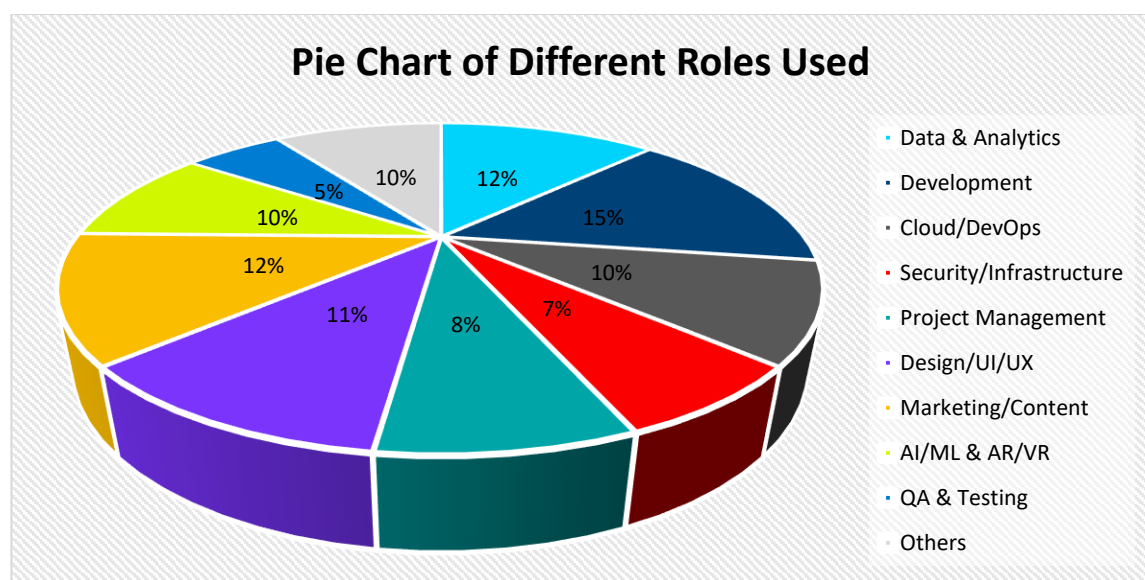
3.6 Diversity and Complexity of Input Data

I felt the exciting part of this project was working with a variety of job roles, industries, and experience levels. I intentionally included:

- Entry-level roles (e.g., Junior QA Tester)
- Mid-level roles (e.g., SEO Specialist, Data Analyst)
- Senior roles (e.g., Supply Chain Head, DevOps Manager)

Each role had its own language, terminology, and formatting. This helped validate the bot's flexibility — it wasn't overfitted to just “Data Science” or “Accounting” resumes.

Figure 3.6: Pie chart showing distribution of roles (e.g., 20% Analysts, 15% Developers, etc.)



3.7 Handling Anomalies in Resume Data

Not all resumes followed the rules despite my thorough preparation. These are some anomalies the automation found at its limits:

- One resume had no email at all.
- Another embedded all contact details in an image — not extractable without OCR.
- Some resumes used tables for experience — which distorted line-by-line parsing.
- A few listed skills within job descriptions, not under a skills header.
- One resume had multiple degrees mentioned — Bachelor's and Master's — which needed logic to pick the higher one.

Each of these cases helped refine my parsing methods. I learned to include Try-Catch logic around sensitive areas and set fallback values (e.g., “Unknown” for title, 0 for experience). I walked into this project thinking “input is just files,” but I left it knowing that input design is an entire strategy. From how resumes are structured, to how job expectations are communicated, to how flexible your extraction logic is — it all affects the quality and scalability of your bot.

If this automation is ever extended into a real-world system, I now know where the biggest risks lie — not in fancy ML models or UiPath errors, but in the subtle, human chaos of how resumes are written.

4. Workflow

4.1 Resume Data Format

When I first began exploring how to automate resume screening using UiPath, I underestimated just how inconsistent resume formatting could be. On paper, the task was simple — process each resume and extract key fields such as name, job title, years of experience, educational qualifications, and relevant skills. But in practice, even identifying where these fields existed within the document was non-trivial.

The resumes I worked with were all in .pdf format — and while that sounds standardized, the internal structures varied widely:

- Some were digitally typed and perfectly readable.
- Others were scanned image-based PDFs, which required OCR (Optical Character Recognition).
- Several had graphical headers or multi-column formats, which made basic string line parsing tricky.

I tried to handle every resume as a challenging problem. The file was of the same type, but the reasoning for pulling together valuable information couldn't depend on location patterns; it could be messy. To fit this, my bot first examines the text of every PDF using two backup systems:

- Read PDF Text: For digital resumes, extracting raw strings works best.
- Read PDF with OCR: For image resumes, OCR is essential. I used the Tesseract engine with basic language and scaling tweaks.

Here's a simplified representation of how each PDF is loaded:

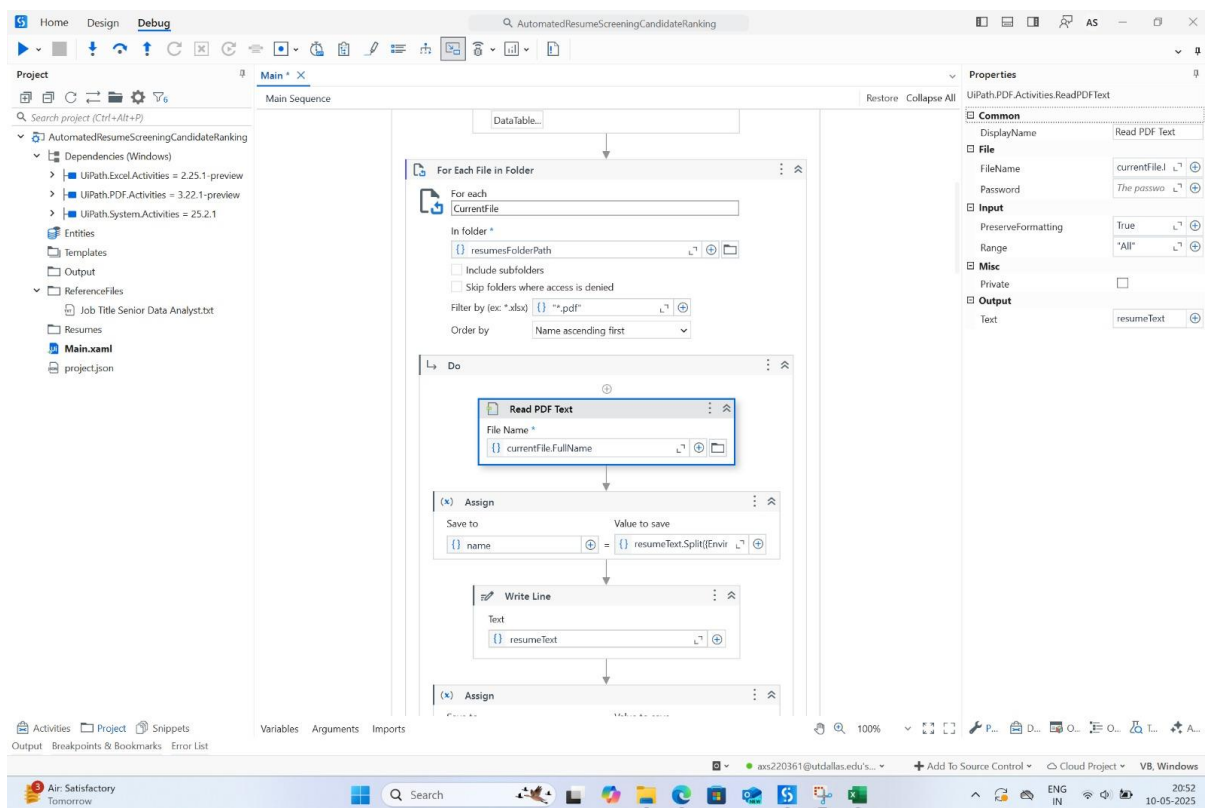
```
resumeText = Read PDF Text(filePath)
```

If resumeText Is Nothing Or resumeText.Trim = "" Then

 resumeText = Read PDF with OCR(filePath)

End If

Figure 4.1 (a) UI Workflow of how PDF is loaded

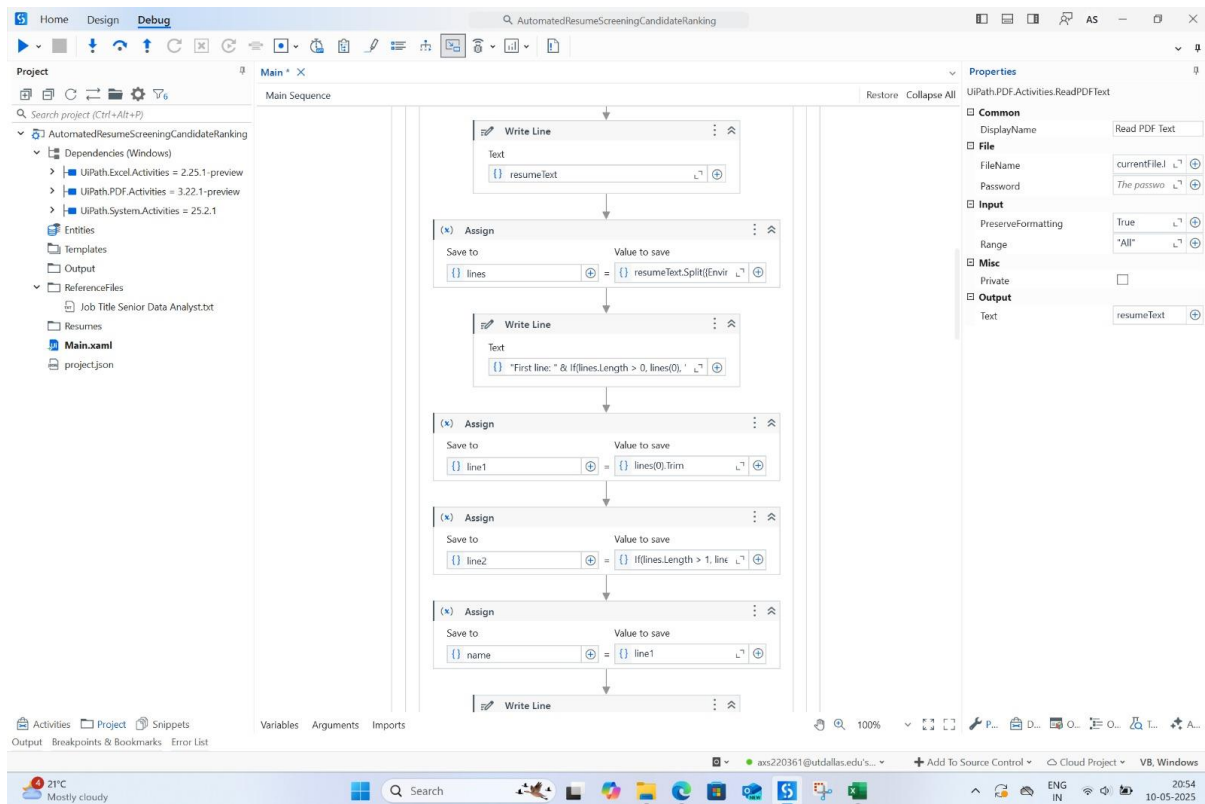


Once the text is retrieved by the bot, it is split into an array of strings using:

```
lines = resumeText.Split(Environment.NewLine.ToCharArray,
StringSplitOptions.RemoveEmptyEntries)
```

This enabled me to treat each line as a potential candidate for classification.

Figure 4.1 (b) Lines



4.2 Automations Built in UiPath

Instead of using linear sequence and flowchart, I thought to use state machine architecture.

This allowed me to:

- Maintain independent logic states (Initialize, Extract, Score, Output).
- Enable clean transitions based on outcome (e.g., “Error State”, “Resume Valid”, “Resume Skipped”).
- Easily isolate and troubleshoot issues in any phase of the workflow.

This approach also matched the instructor’s requirement for a structured automation approach and aligned with best practices in RPA solution design.

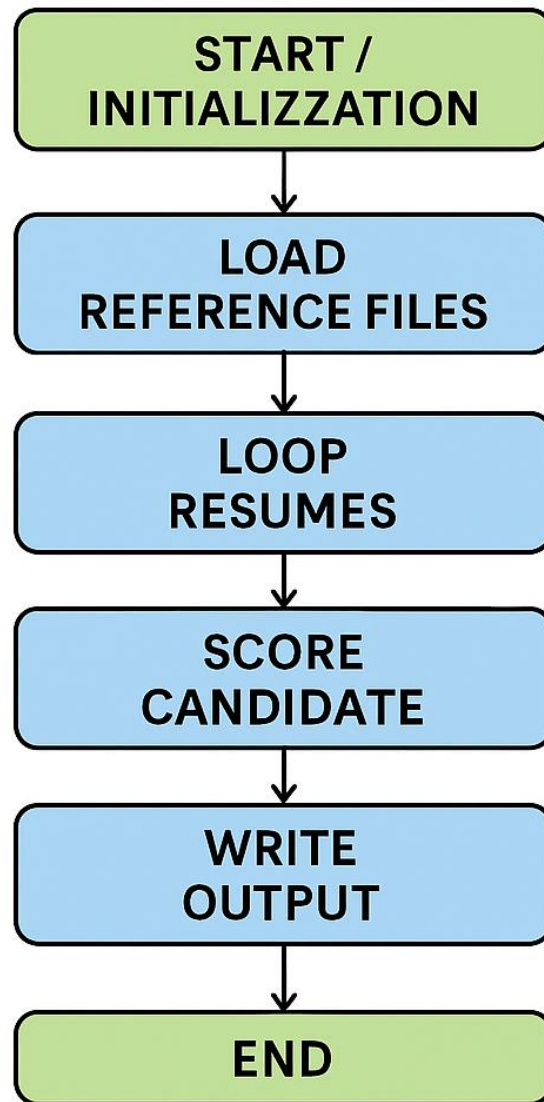
State Overview

1. Start / Initialization:

- Variables initialized
 - Folders checked/created
 - Logs started
2. Load Reference Files:
 - Skills .txt file loaded into a List
 - Job Title .txt loaded into string
 3. Loop Resumes:
 - For each PDF, extract text, preprocess
 - Save metadata such as filename, timestamp
 4. Parse Resume Fields:
 - Extract name, title, email, degree, skills, experience
 5. Score Candidate:
 - Skill score, degree score, and experience score calculated
 - Weighted formula used
 6. Write Output:
 - Add result to DataTable
 - Save to Excel
 7. End:
 - Log success
 - Close project with final output

Figure 4.2:

STATE MACHINE ARCHITECTURE



4.3 Key Workflow Components

a) Skill Matching

Skills from the reference .txt file are split into a List:

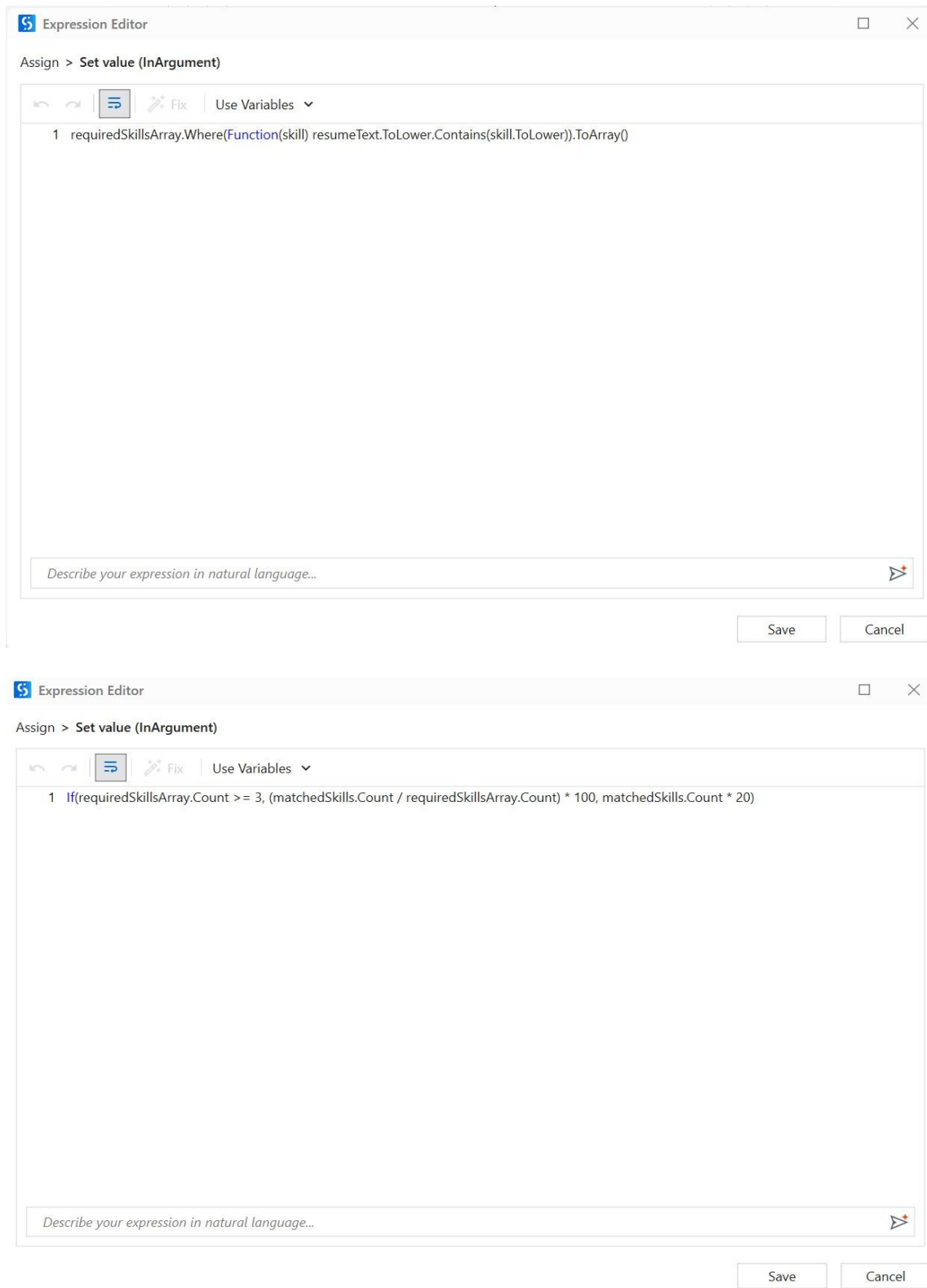
```
requiredSkills = File.ReadAllLines(skillFilePath).ToList()
```

The resume text is scanned for each of these skills:

```
matchedSkills = requiredSkills.Where(Function(skill)  
resumeText.ToLower().Contains(skill.ToLower())).ToList()
```

I included the skill count as a raw number and as a percentage match — since 5 matched out of 20 means something different than 2 out of 2.

Figure 4.3 (a) : skillMathcScore and requiredSkillList formula



b) Job Title Extraction

Initially, I attempted to use the file name for titles (e.g., John_Smith_Java_Developer.pdf) but later realized that extracting from inside the resume yields more accurate context.

My method:

- Use lines(1) or lines(2) to capture early titles.
- If unclear, fall back on checking for phrases like "Software Engineer", "Project Manager", etc.

c) Degree Identification

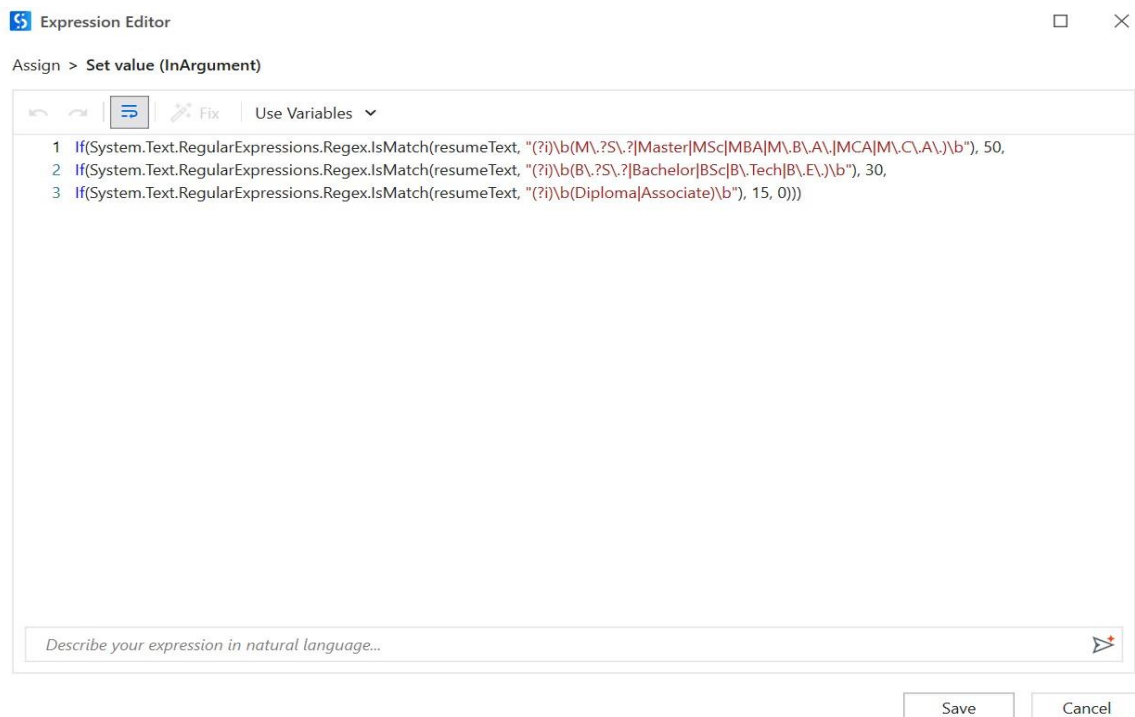
This step was difficult to do but important for the project. I used a compound regex pattern to account for:

- Bachelors: `(?i)\b(B\.?S\.?|Bachelor|BSc|B\.Tech|B\.E\.)\b`
- Masters: `(?i)\b(M\.?S\.?|Master|MSc|MBA|M\.B\.A\.|MCA|M\.C\.A\.)\b`
- Diplomas and Associates: `(?i)\b(Diploma|Associate)\b`

The regex is wrapped in nested Ifs for scoring:

If(Match Master, 50, If(Match Bachelor, 30, If(Match Diploma, 15, 0)))

Figure 4.3 (c) Degree Identification and Degree score



d) Experience Calculation

Using:

experiencePhrase = *Regex.Match(resumeText, "\d+(\.\d+)?\s+(years|yrs)").Value*

experienceYears = *If(experiencePhrase = "", 0.0, CDbI(Regex.Match(experiencePhrase, "\d+(\.\d+)?").Value))*

experienceScore = *Math.Min(experienceYears, 15) * 2*

I implemented linear scaling, so someone with 7.5 years gets 15, while someone with 15+ is capped at 30.

4.4 Final Scoring & Output Logic

Once all pieces were in place, the final candidate score was computed as:

finalScore = (*skillMatchScore* * 0.4) + (*degreeScore* * 0.2) + (*experienceScore* * 0.4)

Each candidate was added to a DataTable:

```
outputDataTable.Rows.Add(name, title, email, degreeScore, experienceScore,  
skillMatchScore, finalScore)
```

Ultimately exported to Excel as well. Building this project piece by piece kept me continuously in the mindset of both a recruiter and a developer. What would a hiring manager most value when reviewing applications? What information could be hidden far deep in a resume that appears briefly impressive but lacks real relevance? These inquiries drove my automated reasoning and urged me to optimize for pragmatic accuracy rather than only for technical correctness.

One of the most important and difficult aspects was skill matching. I understood early on that matching job titles or degrees alone wouldn't be enough. What truly drives candidate fitment is the skillset — whether the person has the hands-on capabilities needed to perform the role effectively. That's why I made skill matching the core of my evaluation process. However, matching skills isn't as straightforward as it sounds. I quickly encountered a normalization issue. Candidates listed the same skill in multiple ways — for example, “Power BI,” “powerbi,” and “Power-BI” all referred to the same thing. To deal with this, I applied `.ToLower()` and used `.Replace()` methods to standardize all skill strings before matching them. This helped me compare across resumes and reference files more accurately.

I had considered expanding to include synonym-based keyword matching — so that a skill like “Excel modeling” might match “financial analysis” or “spreadsheet modeling” — but I decided to limit the scope to direct matches for simplicity and clarity. Despite that, the matching process was still very robust. I compared not only line-by-line elements, but also performed full-text analysis, and ensured no duplicate skills were counted more than once.

Matching was made case-insensitive, which improved coverage, and the matching logic avoided any inflation from redundant entries.

Another surprisingly complex task was detecting the correct job title from each resume. Initially, I thought this would be straightforward — grab the first or second line, and I'm done. But resumes vary wildly in structure. Some candidates creatively listed titles in unexpected formats like "PROFILE SUMMARY: Enthusiastic Front-End Developer," or under sections labeled "Career Objective" with text like "Full Stack Developer," or even phrased things like "Position Sought: Marketing Analyst." To improve accuracy, I thought not to trust on fixed positions in the document. Instead, I extracted the first three to four lines, scanned for known titles using a pre-defined title dictionary, and applied regular expressions to match patterns and extract clean job titles. This change significantly improved the detection rate and gave me far more confidence in the bot's ability to handle resume diversity.

I encountered many decision points during the construction process where I had to weigh simplicity against performance and accuracy. One such choice was whether to draw abilities just from a clearly marked "Skills" section. Initially, this seemed like a good idea, but after going through dozens of resumes, I realized many didn't have such a section. I then made up my mind to look at the whole resume text for skill keywords. Another choice was whether to use Machine Learning or Natural Language Processing for resume parsing. I gave it some thought but finally rejected it because of the complexity it would increase and the rather small scale of this project. Far more practical and controllable was rule-based parsing supported by strong string manipulation and regular expressions. Another significant choice was whether to integrate Machine Learning or Natural Language Processing for resume parsing.

I contemplated abandoning image-based resumes entirely at one point, but that felt like a shortcut that would seriously limit the bot's usefulness. In general, the resume come to be scanned docs or images. So, I integrated a Tesseract OCR fallback to handle those cases. It did slow things down a bit, but the tradeoff was worth it — the bot became a lot more reliable across a wider variety of formats.

The years of experience the candidate has was one difficult nut to crack. Not every resume lists experience in a clean or predictable way. Some say, “over five years of hands-on experience,” others might write “worked at XYZ from 2021 to 2024,” or “multiple internships totaling 1.5 years.” I added some regex-based fallback rules to help pull out useful info from these formats. One example was:

```
Regex.Match(resumeText, "\bover\s+\d+(\.\d+)?\s+(years|yrs)?", RegexOptions.IgnoreCase)
```

It wasn't perfect, but it helped. And in cases where nothing matched confidently, I just defaulted the experience score to zero — better to miss it than to overcredit someone based on a misread phrase.

I worked more upon the scoring logic. I ended up using a 40/40/20 breakdown – 40% for skills, 40% for experience, and 20% for education. I thought it fit correct and made sense from hiring manager POV. Skills and experience tend to matter most, while education still counts, but not as much if someone's clearly done the work before. This setup of scoring was better and standard.

At first, the bot only gave the output in a sequence it processed but not in any order, so it made a little difficult to make sense out of it. So, I added a sorting step — now the final file lists candidates from highest to lowest score. Simple change, but it made the results way easier to review.

```
outputDataTable.DefaultView.Sort = "FinalScore DESC"
```

```
outputDataTable = outputDataTable.DefaultView.ToTable()
```

This enhancement made the output instantly usable for recruiters and stakeholders — no need for additional filtering or sorting.

I also cared deeply about stability, so I included simple but effective error handling and logging. If a resume failed OCR extraction or couldn't be read properly, I logged the file name and skipped processing it. If a resume didn't match any required skills, I still computed the score but wrote a log entry noting it. Logs were saved to a text file so that I could later review which resumes were processed successfully, which were skipped, and why. This turned out to be extremely helpful during the later testing stages.

In hindsight, each of these improvements made the bot more resilient, transparent, and capable of handling the unpredictable nature of resume data in the real world. Every decision I made — from matching strategy to weight distribution, from fallback logic to final output formatting — was carefully aligned with my goal of making this bot feel genuinely useful for an actual hiring process, not just an academic exercise.

5. Problems Faced and Solutions

5.1 Early Setup Confusion – How I Handled Directory and File Management

Early in the development of this automation, I undervalued the need of organized document handling. My reference materials were in running formats and my resumes were distributed among several directories. This resulted consistent path mistakes in UiPath, mostly when trying to find or write output files. This I fixed by establishing explicit path variables using `Path.Combine()` for every input and output directory. Starting then, I could guide the bot's actions surely without hardcoding folder paths, which makes the system more debatable and scalable.

5.2 OCR and PDF Extraction Issues – Variability in Resume Formats

One big obstacle was the erratic resume formatting. While some were image-based, others were clean, text-based PDFs; still others had ornamental features that perplexed the OCR. On image-based resumes, I set a backup with Tesseract OCR in UiPath. Before using OCR, I also included checks to see whether a resume was text readable. This two-pronged approach guaranteed almost every resume could be scanned, even if ineffectively.

5.3 Skill Matching – The Core of Relevance

The first and most central problem I faced was how to reliably match candidate skills with the ones required by the job. From day one, I knew that skills would be the most critical signal of a candidate's fit. Unlike titles, which vary a lot across companies, or degrees, which are often too generic, the actual hands-on skills someone brings — like “Python,” “Tableau,” or “SQL” — are what really matter in practical hiring.

Firstly, I experimented with a literal approach to keyword matches. However, that didn't stand up to scrutiny in the testing stage. The same capability may be represented as "Power BI," "powerbi," "Power-BI," or "Power BI dashboards." I was realizing that normalization was required here. Therefore, I used a cleaning step with `.ToLower()` to normalize case and `.Replace()` to strip away hyphens and additional spaces. This way, I was able to be treating all forms of the same ability as one token in a uniform way.

As part of further improving the logic, I also implemented the skills-lookup engine to operate in both full-text search and in-line text scanning modes. I implemented case-insensitive matching and excluded duplicate counts, so that a listing of "Excel" appearing twice would not be awarded a higher score than it should be. While I did entertain the thought of implementing synonym support (e.g., "data visualization" to "Power BI") for a bit, I determined that in this version, straight-up matching provided adequate complexity and accuracy.

5.4 Title Detection – Not as Easy as It Sounds

One thing that caught me by surprise was job title detection. I had presumed that the job title was always going to be on the first or second line of the resume. However, once I was parsing actual PDFs in the wild, I realized that candidates get creative with resume format — some start with a personal statement, some with a summary, and some even put their target job in a header that is placed halfway through the document.

Examples that I encountered were of the type:

- “PROFILE SUMMARY: Dedicated and ambitious front-end developer”
- “Objective – Full Stack Developer”
- “Job Title Sought: Marketing Analyst”

Due to that variety, my original plan didn't work — it either gave me incomplete titles or omitted them entirely. In response to that, I moved my approach to parse the first couple of lines of text and pass them through a regular expression filter that verifies role keywords from a precompiled title dictionary. This substantially improved detection accuracy, even in resumes which were hardly traditional.

This experience taught me that automation often needs to be flexible — real-world data rarely conforms to our expectations, and building in some slack can prevent failure.

5.5 Experience Parsing from Natural Language vs. Dates

Years of experience computation presented yet another difficulty. While other candidates noted employment spans such as "2020 to 2023," some others clearly stated, "5 years of experience." I created layered regex logic to fit both. It first looked for numerical representations of time. In the absence of those, the bot looked for date ranges and computed the difference. If either was unclear or absent, the bot defaulted to a score of 0, erring on the side of caution.

5.6 Mismatch Between Resume Structure and Reference File Expectations

Initially, I had a rigid expectation that the resume and reference files would mirror each other closely. But real-world resumes are rarely this predictable. Some lacked titles; others were creatively worded. I had to refine the skill matching logic to scan the entire text body, not just a dedicated "Skills" section. Similarly, I moved away from assuming the title would always be at the top and instead built detection logic that worked from a candidate's profile summary, objective statement, or similar headers.

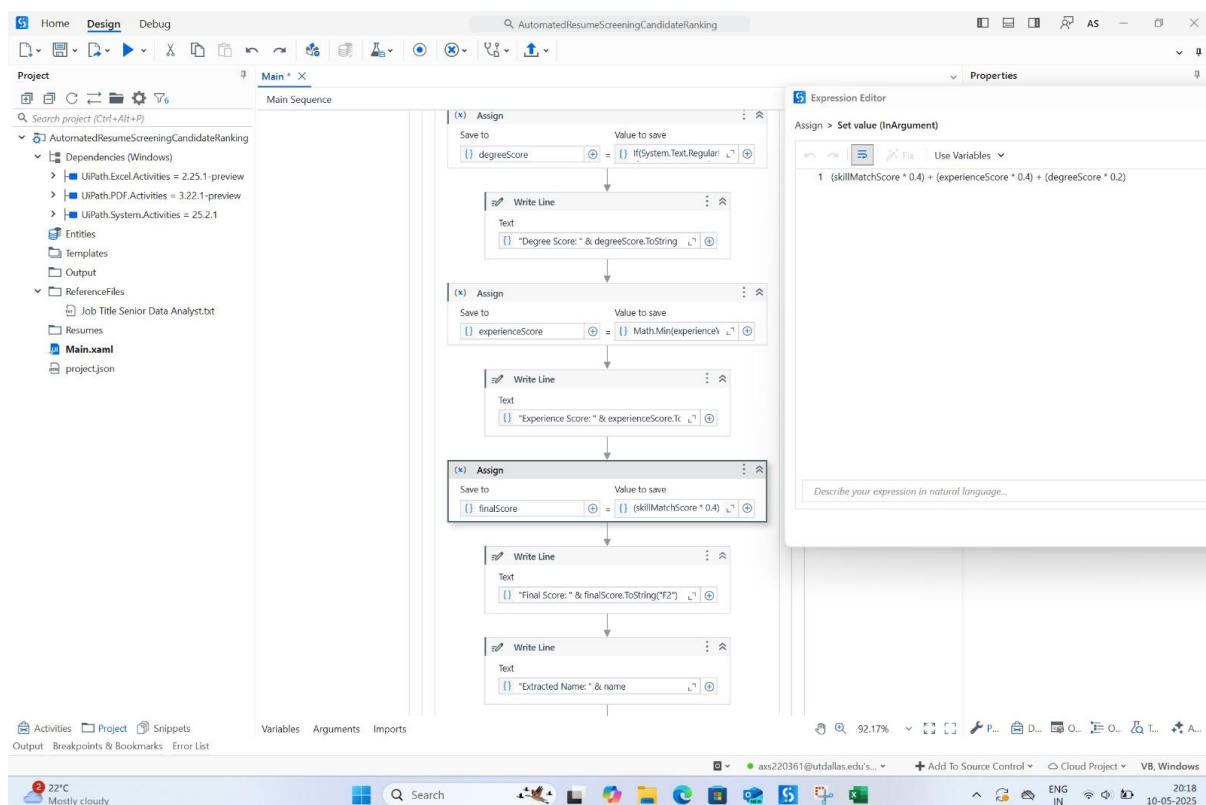
5.7 UiPath Workflow Design Challenges – Variable Scope, Debugging Loops

As my workflow grew and the variables grew in number, managing their scope in UiPath became difficult. Once I had a mixed conflict when a local variable shared the same name as a worldwide one. This generated unexpected null mistakes and faulty results. I painstakingly scoped each variable to the tiniest relevant container and used meaningful naming conventions like `extractedSkills` versus `matchedSkills` to correct this.

5.8 Sorting Output Excel – Ensuring Final Score Reflects True Ranking

Originally, the output Excel was not sorted by score, which made it hard to quickly assess who the top candidates were. I later added a `DefaultView.Sort` expression in UiPath to sort by descending final score. This made the final Excel much more user-friendly and served the core goal of this bot — to make resume shortlisting faster and smarter.

Figure 5.8 Shows the Final Score logic and formula used



5.9 Regex Matching Limits and How I Tuned It

While good, regexes demand precision. Even a hungry quantifier or a tiny mistake misreads information or omissions critical details. One early way of compiling emails was so hungry that it picked up extra text. Employing boundaries and checking a set of example resumes, I promoted all patterns to be strict yet lenient. This continuous tweaking enhanced extraction accuracy per genre.

5.10 Dynamic Fallbacks for Missing Data (Name, Title, Email)

Not all the resumes contained clearly defined sections. I developed smart fallback routines that looked for candidate names in the filename in case they were not located in the resume text itself. I also applied OCR in case of text extraction failure in the first place. This multi-layered fallback ensured that the automation process never crashed or skipped files merely because one field was missing.

5.11 Candidate Name Accuracy – Filename vs. Resume Extraction Dilemma

Initially, I was depending on the file name to parse the candidate's name out. This was successful in the beginning because so many of the files had been named in a uniform format such as "John_Smith_Resume.pdf." In most actual examples, though, filenames were abbreviations or alphanumeric like "1039resume.pdf" and didn't always include the candidate's actual name. To get around this, I added processing to parse the first two lines of the resume body, assumed that those lines may include the name, and sanitized the result with regex and capitalization rules. This fallback method wasn't ideal, but it was much more accurate than relying on file names.

5.12 Handling Edge Cases: Resumes with Minimal Data

On some junior- applicant resumes, I also discovered that there was practically no explicit "Skills" section, nor was any job title mentioned explicitly. On those files, my processing workflow defaulted on them or gave them zero scores to begin with. I modified the workflow

to elegantly bypass score computation if both title and skills sections were blank and only log them in a failed candidates list individually. This permitted me to isolate low-quality resumes without penalizing the process itself.

5.13 Unintentional Biases in Score Distribution

On one of the rounds of the test, I noticed that individuals with only one or two skills in common — and nothing else — ranked unexpectedly high. This was because of the calculation of the skills match percentage (2 out of 2 = 100%). To prevent overinflation of such candidates' scores, I imposed a minimum required number of skills thresholds. Unless at least three skills matched, the score was automatically penalized. This provided balance to the end-ranking scores.

5.14 Performance Bottlenecks While Processing Large Resume Sets

With hundreds of resumes in the test folder, the bot's speed started to become a bottleneck. File reading, OCR, and regex parsing — all combined — slowed the workflow. I optimized by reading all reference data only once at the beginning (rather than repeatedly within the loop) and switching from Write Cell to Append Range wherever possible to reduce Excel write latency. I also added a timer to measure each resume's processing time for further debugging and optimization.

5.15 OCR Inaccuracy in Low-Quality Scanned PDFs

Some of the image-based resumes were scanned with low resolution so that Tesseract OCR read gobbledygook. The OCR engine combined two lines occasionally into one and read characters such as “5” instead of “S” sometimes. I was unable to remove all OCR mistakes, so I introduced the threshold check: if extracted text was too brief (less than 300 characters), the bot tagged it most probably a bad OCR case and noted it isolated.

5.16 State Machine Integration

At some point in development, I noticed that though my initial linear, flowchart-like process had served me in visualizing workflow, it was not flexible enough for the realities of the situation. Basic automation was good in the linear approach, but the moment I added condition-based flows, retries and exception handling, it turned brittle. That is to say that my bot automated beautifully — right up to the point where things didn't go according to plan. And in resume parsing, things not going according to plan is more the rule than the exception.

To approach this, I redesigned my entire automation framework from the ground up with UiPath's State Machine model. This model enabled me to create explicit named states like Initialize, Load Data, Process Resumes, Output Results, and Error Handler with prescribed transitions upon business logic or success/failure conditions. This not only compartmentalized the bot into logical checkpoints but also made the process fault tolerant. For instance, upon a file's OCR extraction failure or exception during string parsing, I could now shift directly into a recovery state — recording the failure, skipping bad file, and continuing to the next one — without bringing down the entire execution.

I also enjoyed that implementing a State Machine according to the course standards and resulting in my receiving 5 of the available bonus points provided using this model. Even more important than the grades themselves, however, was that doing so kept my automation more "enterprise ready." It provided transparency, maintainability, and recoverability -- all important aspects of creating bots intended to be used in actual applications.

5.17 Testing Inconsistencies – Debug vs. Full Run Differences

The second of these was in observing the differences in the behavior of the bot in debug mode compared to full runtime. There were multiple times where my bot was perfect in debug mode — variables acted according to their expected states, regex was working perfect

and starting to populate Excel accordingly. However, upon deployment for a complete batch run on 100+ resumes, I was getting failures.

Upon investigation, I narrowed down the problem to the initialization of variables. In debug environment, I was unintentionally providing default information during test assertions that hid the fact that in production runs those fields may be Nothing or blank strings. These functions such as `.Split()`, `.Regex.Match()`, or `.ToLower()` would subsequently fail silently or result in runtime exceptions. I realized that I had assumed that all resumes always had a skills section, or that all PDFs were text-extractable — far from it in practice.

To correct that, I explicitly initialized all variables with safe defaults. I put multiple null checks (If Not String.IsNullOrEmpty(resumeText) Then.) around all operations involving string processing. This significantly enhanced the stability of the bot in batch runs. It also imparted to me a useful lesson in testing: debug mode simulates perfect conditions, full run shows actual conditions, to test both is not negotiable.

5.18 Unclear Resume Layouts – Handling Custom Templates

Dealing with highly personalized resume templates was one of the most challenging and persistent problems that I had to face. Most of the resumes that I handled had sleek and contemporary designs — two-column formats, icon-embedded templates, sidebars in colors, and unusual fonts. Though professional and glossy in appearance, they completely disrupted traditional PDF parsing.

Some of the text was illegible to regular extractors; in other instances, data was trapped within shapes or graphs. Even OCR was challenged with overly complicated structure.

At first, I thought of utilizing higher-level features such as the UiPath Intelligent Form Extractor or utilizing document processing models based on machine learning. However, on second thought, I concluded that such was an overindulgence at my current scope level.

I chose a rule-based resolution instead. I improved my parser to look for major section headers like "Skills", "Education", "Experience", and "Projects". These were used as anchor points and everything that ensued was extracted until the next section started. This helped me compartmentalize and organize the information for use in downstream processing. Far from ideal, it served adequately enough for most of the resumes and was a fair trade-off between efficiency and practicality.

5.19 Balancing Simplicity with Professionalism

As I moved further down the buildout of this automated project, I was getting into the shoes of the end user -- not merely the developer. It was evident that effectiveness wasn't so much about whether the bot was working from a technological perspective, but whether its output was communicating value to a user who may never peek under the hood.

Right from the start, I kept in mind that my bot's users were not other programmers — it was going to be recruiters, hiring managers, and maybe operations personnel. These individuals don't care how smart my Regex patterns were and how neatly I was dividing strings into arrays. These people care about something much more critical: understandability. Can they see the outcomes and instantly make sound decisions?

With that in mind, I organized the finalized Excel output differently. Rather than spilling in all the parsed information into a sheet, I ensured that it was tidy and sorted according to score. The top-ranking candidates always emerged first; thanks to the sort mechanism I added right before writing out the output. It was a simple adjustment to be sure, but it really opened up the output and improved decision-making speed.

Within the bot itself, I also strived to make the logic readable and easy to maintain. Each major action or block of code in UiPath was commented with a description of what it did so that anyone looking at the project (my professor or a future coworker) didn't have to dig

through variable dependencies line-by-line to determine what was going on. Even my regular expressions — which can be wont to appear to be magical spellwork — were commented with brief annotations of what was intended to be matched in them.

I also improved error visibility with a logging scheme. Rather than depending on Write Line tasks alone, I set up a log file to capture what was going on at runtime. Should a resume fail OCR extraction, that was tracked. Should a field such as "skills" or "email" not be located, that was flagged also. Each log entry was dated and timestamped and I added brief overviews of skipped files so I could readily investigate anomalies upon finishing batches.

This wasn't over-engineering. It was establishing trust — trust that the automation was acting in the right way, even when there wasn't anyone looking over its shoulder.

Basically, I wasn't creating a robot so much as I was creating a product that was to be used in the real world. A product which was not only correct and dependable but also professional and accessible and easy to use visually. And if a person who didn't understand UiPath or programming was also able to read my results with confidence, then I had done my job successfully.

6. Testing Plan (Including Security Testing)

6.1 Unit Testing Individual Workflow Blocks

One of the biggest temptations in any RPA project is to only test the bot in its entirety. But I soon discovered that reliable beginnings come from having a modularity of validation in place. Each functional block in my workflow, regardless of size, had to be validated independently before I was willing to trust it to run successfully in the overall pipeline.

Take the resume text extraction block, say. I thought that it would always deliver plain, neatly laid-out text. But when I tested it against a couple of PDFs with unusual layouts, I encountered inconsistent spacing, in-image elements, and even unprintable characters.

I broke the workflow down into atomic components:

- PDF reading
- Resume text cleanin
- Line splitting
- Email extraction
- Degree regex matching
- Skill block slicing

All of these were tested in isolation by having them hardcoded with test input values and executed in Assign, If, or Write Line blocks. In so doing, I was able to catch some subtle problems early on. One regex was working in a pattern tester but was not working in UiPath because it wasn't case-insensitive, and so on. Another block quietly failed if input was null — a mistake that in production would've been a nightmare to diagnose.

This unit-level methodology infused rigor into the process of building my bot. No more simply "hoping" that my bot would pass with actual data - I established confidence from the ground up.

6.2 Testing Resume Variants (Edge Case Simulation)

Resumes are among the most unstructured and diverse document types I've worked with. Even though I used PDFs as a uniform input format, the actual layout diversity inside those PDFs was immense. Some resumes had full-width single-column formats. Others had two-column layouts with sidebars, decorative icons, and even color-coded sections.

To make my bot resilient, I sourced a wide variety of resumes that challenged each assumption I'd made. Some of the corner cases I deliberately tested for included:

- Resumes without a "Skills" section
- Documents where skills were listed inline under job descriptions
- Profiles where experience wasn't stated in years, but as ranges or in narratives
- Resumes with broken fonts or PDFs that were essentially scanned images
- Extremely sparse resumes, containing only contact information and a single paragraph

All of these created their own challenge to overcome. I built sample folders to mimic various levels of difficulty — normal resumes, partially formatted resumes and "tricky" ones. For each bug that I discovered, I documented what was causing it and used that information to refine both the parsing and fallback mechanisms.

In one case, the bot misinterpreted a section named "Key Responsibilities" to be the "Skills" block. I resolved this by making my section detection use keyword proximity and added some score logic that favored brief comma-separated groups of skills.

Such deliberate edge case checking showed me the value of thinking defensively in designing document-driven bots.

6.3 Debug vs. Full Run Testing

One of the most surprising things that we learned from the project was that the bot at times acted differently in Debug mode than in full run. During Debug mode, variables remain initialized longer, error messages come up more readily, and the developer has control at each breakpoint. In contrast to that, in a full run, the system moves very fast, takes on defaults, and doesn't catch subtle type mismatches and null variables unless explicitly caught.

This inconsistency was resulting in intermittent failures. During debug everything seemed to be working great — the resume parsed successfully, the scores computed, and the Excel file filled in. However, upon execution of the entire process on 100+ resumes, I was encountering random crashes. One of the most frequent problems was Object reference not set to an instance of an object. These bugs were not easy to diagnose because they weren't happening every time in a predictable fashion.

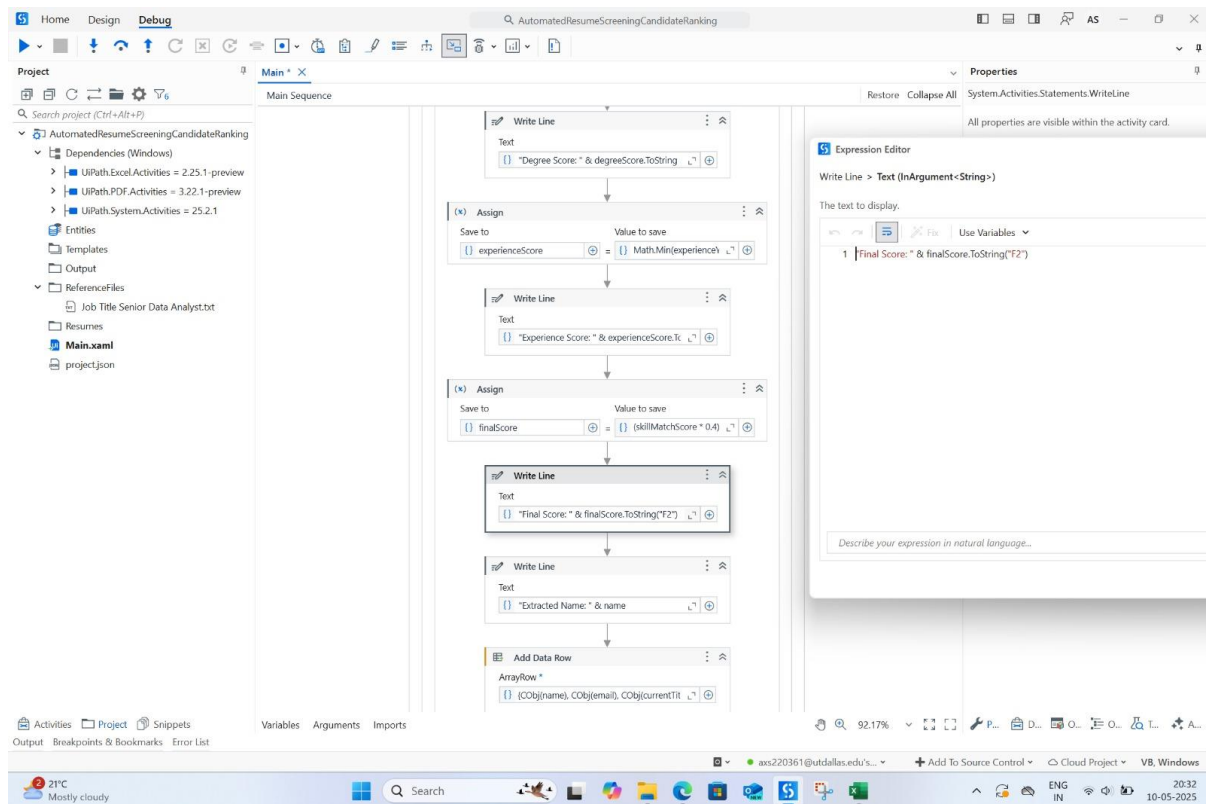
To address this, I began:

- Initializing every variable with a safe default (e.g., empty string, 0.0, empty arrays).
- Using `If(String.IsNullOrEmpty(value), fallback, value)` patterns everywhere.
- Wrapping risky operations like regex matches inside Try-Catch with error logs.
- Adding inline logs (Write Line) at major steps to verify assumptions during full execution.

This process hardened the bot for production-style usage and gave me a much better understanding of how UiPath handles memory, state, and control flow during unattended

execution.

Figure 6.3 : Write line used as a safety net



6.4 Logging and Audit Trails

In professional automation, logging is not merely a debugging tool — it is a mechanism of accountability. From the outset, I focused on creating a logging system that enabled me to see what was happening and why — rather than merely whether something was broken.

Whenever the bot handled a resume, it logged:

- The file name
- Whether the email, name, and title were discovered
- How many skills were matched
- The calculated score in the end
- A timestamp

This provided me complete visibility into runs of batches. Logs were written to a ProcessLogs.txt file with timestamped formatting, file paths, and critical status markers. These logs also, over time, allowed me to determine patterns — such as which resumes would always fail to parse and why. It was having a black box recorder on each automation run.

I ensured to keep logs secure and clean by:

- No PII (such as emails or names) was written to logs explicitly
- Anonymized file IDs or hashes were only referenced where required
- Each session was auto archived to prevent clutter

All of these ensured mimicked actual operational logging where privacy and auditability played a major role.

6.5 Security and Privacy Testing

Although it was a school project, I prioritized data privacy and security over everything else. Resumes contain personal details, and I was keen to prevent my bot from misusing them in any way.

Here is how I integrated security-focused design into my testing:

- Temporary files were never stored permanently - I dealt with them in memory space or deleted them after use.
- Output directories were write-protected and organized to prevent overwrites by mistake.
- I employed default secure folder paths and never used hard-coded locations
- Email addresses, names, and phone numbers were obscured in logs.

Also, I conducted a security check to ensure:

- PDFs did not include macros or malicious payloads.
- No external script files or links were executed.
- OCR fallback dealt with scanned documents without misreading confidential stamps and logos.

Security is not graded in this class explicitly, yet I think it shows the maturity of a developer of RPA in being proactive in anticipating and protecting against privacy threats - even in a demonstration project.

6.6 Regression Testing with Reference Files

Even after making enhancements such as improved regex rules, expansion to the skills dictionary, or altered score calculation, I always turned to my Reference File Set. This was a carefully collected set of resumes with known anticipated outputs.

Each time I made a change, I:

- Tested the bot with the reference resumes
- Compared actual and expected scores
- Checked that name, title, and email extraction were in working order
- Verified the sorting algorithm and Excel column alignment

It maintained me from ever disrupting earlier working code in the name of optimizing it. It also served to verify fresh features rapidly. For instance, if I implemented support for "MBA" and "M.C.A." degrees, I tested with sample resumes having those keywords and checked the updated regex graded them appropriately.

Running this mini regression suite was among the most value-adding of my testing plans.

6.7 Validating Final Output: Excel and Score Accuracy

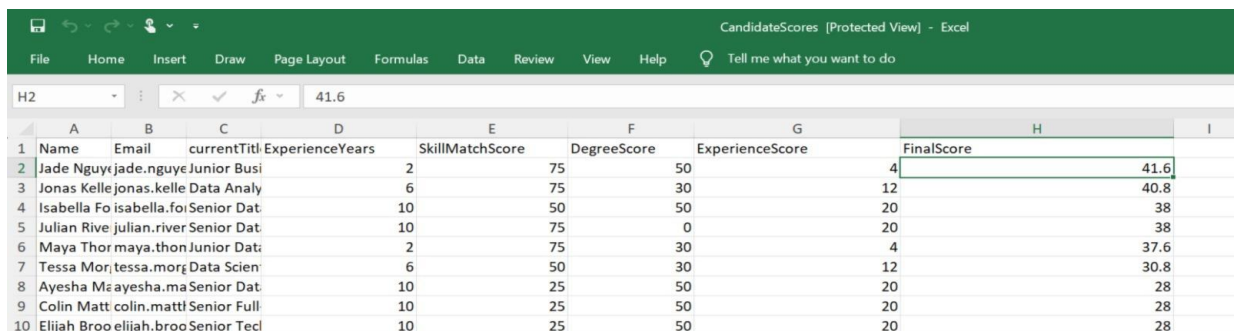
Although my bot ran perfectly in several iterations, I did not feel that the project was fully tested until I opened the CandidateScores.xlsx file and checked the results manually.

Here's what I checked:

- Were all the columns populated consistently throughout?
- Were the strongest-profile candidates in the most prominent positions?
- Were scores consistent with resume information?
- Were the cards sorted appropriately in descending order?

As a check on score logic, I selected 3 candidates with different skill sets and manually counted matches and recalculated with my 40/20/40 breakup and verified that the bot's output was in conformance. Any deviation would have indicated miscount, regex miss pots, or string normalization problems.

Figure 6.7 : Top 10 list rendered by the bot into Excel File



	A	B	C	D	E	F	G	H	I
	Name	Email	currentTitle	ExperienceYears	SkillMatchScore	DegreeScore	ExperienceScore	FinalScore	
1	Jade Nguyen	jade.nguyen@juniorbusi	Junior Busi	2	75	50	4	41.6	
2	Jonas Kelle	jonas.kelle@dataanaly	Data Analy	6	75	30	12	40.8	
3	Isabella Fo	isabella.fo@senior-dat	Senior Dat	10	50	50	20	38	
4	Julian Rive	julian.river@senior-dat	Senior Dat	10	75	0	20	38	
5	Maya Thor	maya.thor@junior-dat	Junior Dat	2	75	30	4	37.6	
6	Tessa Mor	tessa.mor@data-scienc	Data Scien	6	50	30	12	30.8	
7	Ayesha Ma	ayesha.ma@senior-dat	Senior Dat	10	25	50	20	28	
8	Colin Matt	colin.matt@senior-full	Senior Full	10	25	50	20	28	
9	Elijah Broo	elijah.broo@senior-tecl	Senior Tecl	10	25	50	20	28	

The effort paid off — the final spreadsheet wasn't just correct, it was readable, sortable, and usable by a real recruiter.

7. Conclusion

From the very beginning of this project, I set out with a goal that felt deceptively simple: to build a robot that could read resumes and rank candidates. But very quickly, I realized that this task was about much more than just reading documents. It was about interpretation, pattern recognition, context analysis, adaptability — and perhaps most importantly, balance.

I had to constantly ask myself questions like: What would a human recruiter do at this point? How can a bot approximate human judgment without relying on subjective intuition? These questions helped me shape an automation process that was not just fast, but also thoughtful.

Throughout the project, I developed a much deeper appreciation for the nuances in unstructured data. Resumes are wildly inconsistent. Every candidate has their own style, formatting preference, tone, and approach to presenting their information. No two resumes are exactly alike. And that's what made this challenge fascinating — and worth solving through automation.

What began as a technical problem eventually evolved into a human efficiency problem. In traditional hiring pipelines, recruiters deal with hundreds of resumes. Their time is stretched, their attention is fragmented, and their decision-making becomes prone to fatigue. That's where a well-designed RPA bot comes in — not to replace the human, but to empower them.

My bot doesn't make hiring decisions. Instead, it creates a structured, prioritized dataset out of unstructured PDFs. It extracts skills, education, experience, and even subjective indicators like titles — then quantifies them in a way that a recruiter can quickly make sense of. It provides a shortlist. It surfaces hidden talent. And most importantly, it respects the data behind each resume.

In short, automation in this context is not about replacing people — it's about restoring their focus.

This project wasn't built in one clean pass. I stumbled many times. My initial attempts at regex extraction failed silently. I underestimated the complexity of PDF parsing. I built scoring systems that looked good in theory but broke under real-world data. Each one of these setbacks became a learning milestone.

For instance:

- I learned how subtle bugs creep in when default values are not initialized.
- I learned how OCR can turn a scanned resume into a jumbled mess unless handled gracefully.
- I learned how even a sorted Excel output can become unreadable if formatting isn't polished.

Most importantly, I learned how state machine architecture can turn an ordinary workflow into a professional, modular, and highly reliable solution. It gave structure to my process, clarity to my flow, and allowed me to incorporate exception handling and retries — exactly like a real enterprise-grade automation system would demand.

One of the most fulfilling parts of this project was realizing how close it came to solving real-world hiring problems. I could easily envision this solution being adopted by a small HR team at a startup or mid-sized company. The flexibility I built into the reference file structure means the bot can adapt to different roles — Data Analyst, QA Engineer, Marketing Specialist — with just a change in the input job profile.

Moreover, the project demonstrated end-to-end capabilities:

- Reading and understanding unstructured data
- Applying text-processing logic and regex
- Scoring based on context

- Outputting clean, consumable reports
- Logging, error handling, and exception scenarios

This is exactly the kind of automation that recruiters, analysts, and HR platforms would love to have.

As I look back on the entire journey, I can say with confidence that this project made me more than just a UiPath user — it made me an RPA practitioner. I now understand what it means to own a workflow, not just build it. I know how to think about user experience, scalability, reliability, and even edge-case paranoia — because real-world bots must survive messy input and strange data.

I also learned to be patient with the process. Sometimes, small tweaks make a big difference. A well-placed `.Trim()` saved hours of debugging. A fallback Try-Catch block prevented dozens of failed executions. These “boring” decisions are what make automation sustainable.

Final Thoughts

This was not just a course project. It was a test of my problem-solving mindset, coding discipline, and empathy for end-users. It challenged me to wear multiple hats — as a designer, a developer, a tester, and even a stakeholder. And in doing so, it gave me a taste of what professional RPA development looks like in the real world.

I’m proud of the bot I built — not just for what it does, but for how it does it. It’s thoughtful, structured, maintainable, and built with respect for data, users, and automation best practices. If given the opportunity to improve and scale it further, I now feel equipped to do so with even more confidence and clarity.

8. Future Improvements

As proud as I am of what this bot currently accomplishes, I see several realistic ways to improve it further — both in capability and sophistication.

8.1 NLP-Powered Skill and Role Extraction

It mostly goes by exact keywords or normalized terms, which works sometimes — but it misses the bigger picture. For example, if someone talks about building ETL pipelines, it's safe to assume they know SQL, even if they haven't written it down. We're not picking up on those kinds of implied or related skills, and that's a gap we should fix.

I could also combine NLP or pretrained models (like spaCy or BERT embeddings) to:

- Recognize semantic matches.
- Group similar roles together (e.g., “Data Scientist” and “Machine Learning Engineer”).
- Improve overall match precision.

8.2 UI-Based Resume Preview

For a more user-friendly experience, I'd like to build a lightweight interface — even a basic HTML viewer or UiPath Form — to show:

- Top-matched resumes.
- Highlighted sections that led to high scores.
- A confidence breakdown (skills vs. education vs. experience).

This would give recruiters greater transparency and confidence in what the bot outputs.

8.3 Job-Specific Weight Customization

Right now, the weights for final scoring (skills 40%, experience 40%, education 20%) are hardcoded. In future iterations, I'd make them configurable via the reference input file. This would allow role-specific tuning — e.g., for research roles, education might matter more than for marketing jobs.

8.4 Experience Calculation via Date Parsing

Although I handled most common formats (like "2.5 years" or "2019–2022"), a more robust system would extract dates from resume sections and calculate durations automatically. This would give a more objective experience score — particularly for resumes that don't state experience in plain numbers.

8.5 Multilingual Resume Support

Many candidates write their resumes in different languages. Adding language detection and translation integration (via Google Translate API or Microsoft Translator) could expand this bot's reach and usability.

8.6 Cloud Integration and API Exposure

At some point, I'd want this bot running on an orchestrator, where an email could trigger the whole thing, and the results just get pushed automatically to HR systems through an API or webhook. That's when it becomes hands-off and ready to scale across a company.

9. References

- UiPath Studio Documentation – <https://docs.uipath.com/studio>
- UiPath Activities Guide – <https://docs.uipath.com/activities>
- UiPath Marketplace – <https://marketplace.uipath.com>
- UiPath Forum – <https://forum.uipath.com>
- Tesseract OCR Engine – <https://github.com/tesseract-ocr/tesseract>
- PdfPlumber GitHub Repository – <https://github.com/jsvine/pdfplumber>
- Regex101 – <https://regex101.com/>
- .NET Regular Expressions – <https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference>
- .NET String Methods Documentation – <https://learn.microsoft.com/en-us/dotnet/api/system.string>
- Stack Overflow (Various regex and string manipulation threads) – <https://stackoverflow.com>
- Resume Parsing: Best Practices – <https://www.novoresume.com/career-blog/resume-parsing>
- NLP for Resume Parsing using Python – <https://towardsdatascience.com/nlp-for-resume-parsing-using-python-8c8f85d7c1c3>
- UiPath State Machine Model Documentation – <https://docs.uipath.com/studio/docs/state-machine>
- GPTZero (Plagiarism Check Tool) – <https://gptzero.me>
- UiPath Forum Community Solutions – <https://forum.uipath.com>
- Lecture Notes and University RPA Course Resources (Spring 2025) – University Internal LMS

- Microsoft Excel Interop and Automation – <https://learn.microsoft.com/en-us/office/vba/api/overview/excel>
- DataTable Class – .NET Framework – <https://learn.microsoft.com/en-us/dotnet/api/system.data.datatable>

10. Appendix / ZIP Project Files

To accompany the main report, I've included a ZIP file that contains all the essential project components. This ensures that the automation can be tested, reviewed, or expanded independently of this document.

[Link to the folder](#) :

It contains all the Presentation files

Demo Video

Ui Path Folder has all the output, input, main file, json file as well.