CS 224G Individual Project Summary

I truly think that most upper-level CS classes at Stanford should be structured like CS 224G. Without having additional problem sets or midterms in this class to distract me, I was able to focus solely on the project. Additionally, the independence offered in this class made working on the project especially fun. Purely out of enjoyment rather than necessity, I probably spent more time on this project than any other CS project I've had at Stanford. I came into the class with a problem I'd had at the back of my mind for a while. The last two summers, I did internships at companies that don't have great team matching processes. I got placed on teams that one, I had no relevant experience for and two, didn't have any interest in. This made my summers unenjoyable since I felt like I had to play catchup the entire time. That would've been fine since most learning usually happens on the job anyways, but I had no interest in the project I was working on. This meant I had to spend countless hours working on something that I hated.

I always wondered why we couldn't automate parts of the team matching process. With advances in embeddings and NLP, there had to be a way. \ So, when I heard about CS 224G, I thought matching projects and resumes would be a great project to work on.

One of the biggest insights I gained from this class is something that applies to building any kind of software. When you have an idea for an algorithm to use within an app, start out simple. Implement the most basic version of your algorithm as a starting point and then add complexity from there. It doesn't matter if your initial algorithm doesn't work that well. In my opinion, it's easier to make your algorithm more complicated once you have a working app than it is to develop an app around a super complicated algorithm from the get-go. At the beginning, I spent a lot of time trying to develop a really involved matching algorithm. In the process, I ended up overwhelming myself a little. So, I decided to build a backend that used a very simple embedding/similarity model. I started out embedding entire resumes and project descriptions using the most basic ada-002 embedding model. I obtained matches using straightforward cosine similarities. This algorithm worked horribly to start. However, since we had a working model, it was easy to swap out embedding models. It was also much easier to add complexity to our algorithm and test different ones. By the end, we had an algorithm that was more complicated than any algorithm I had initially envisioned.

We were able to improve our algorithm the most was by intelligently chunking the resume into different experiences. In RAG models, chunking is imperative to finding the most similar documents. In RAG, chunking is traditionally random, but we decided to improve on this. We learned that if you prompt GPT enough, you can get it to chunk a resume into relevant experiences consistently. However, you sometimes have to be scrappy when you prompt. Sometimes, I had to be repetitive in my prompts to make sure GPT actually listened to me reliably. I also had to be very explicit about the format I wanted GPT's output to take. For example, here's the prompt I used to make GPT chunk resumes: f"You are an expert at parsing resumes. I want you to take the following resume text and

divide it into no more than 10 chunks. It is okay to have less than 10, but not more. I want each event in the person's life to be its own chunk, whether it is a project or a job experience. Don't include the name and contact information. Make sure different job experiences and different projects are different chunks, even if they are under the same section. Please make the output a python list, where each element of the list is a string consisting of the text corresponding to the relevant chunk. Make the output a python list that I can readily use in my code. There should not be any text in the output other than this python list. Don't include '''python at the beginning of your output. Do not omit any part of the resume, make sure every line of the resume is included in a chunk. Here is an example of the output I want: ['Experience 1...', 'Experience 2...', 'Experience 3...', etc...]. Here is the resume text: {resume_text}".

All the repetition here was necessary. When I didn't repeat myself ten times, the model always made format errors. The model would also group multiple work experiences together under one "Experience" if they were under the same section, which isn't what we want.

Another key takeaway I had from the project is that there's a real tradeoff between speed and accuracy when using LLMs in an app. You can't just use GPT-4 or the biggest embedding model for everything. This will make the app way too slow. You have to identify the most important tasks and use the best models only for those.

The most useful thing I heard in class was John's prompting advice. John jokingly would say that "Sometimes you have to use all caps and yell at GPT to get it to do what you want." This influenced my own strategy of treating GPT like an idiot and repeating myself over and over. This enabled our app to work.

Overall, building this app was a great experience, and I hope to continue working on it past the class.