Jordan Coleman

10/28/25

Assignment 4 Report

Discussion (as requested):

What prompt you used and why:

For my prompt, I decided to use a large string that I pass into a for loop that loops over the entirety of the dataframe that I am working with. Then, from every entry in my dataframe (that I get from a .csv file) I tell it each Query, as well as it's corresponding Candidate I want it to rank. The prompt is as follows:

```
prompt = f"""
  You are evaluating search results.

  Rate how relavant each candidate is to it's corresponding query on a scale from 0-5.
  0 = not relevant at all, 5 = highly relevant
  Respond with ONLY a number {0, 1, 2, 3, 4, 5}. No words.
  """

for i in range(len(df)):
  prompt += f"""
  {i}:
  Query: {df['query_text'][i]}
  Candidate: {df['candidate_text'][i]}

  """
```

Any variations you tested?

There were no variations that I tested. I only did my input between 0-5 since I figured that was all I needed to do. I figured that would be enough to suffice.

Where the LLM improved or failed:

The LLM overall gave worse results. However, I may need to point out that for some odd reason, whenever I was asking for responses on each query, and whenever I would split those responses into a list of floats, the size of the responses would either come out too large or too small to be fit inside of my Dataframe. It seems that each time I run the same query, the size of the response was always different. To work around this, I test if the length of the responses were greater than the size of the Dataframe, or less than the size of the Dataframe. If greater, than I remove the responses in the back until it matched the size of the Dataframe. If lesser, than I add 0.0 to the responses list until it matched the size of the Dataframe. Pandas would not allow me to insert into my Dataframe unless the list I passed it was the same size as my Dataframe. This may not have been the best thing to do, but it was the best option I could provide.

How cost, latency, or API constraints affected your design:

The program did in fact take a long time to run. I was just introduced to a method to make the program run a bit faster, and that was to have the LLM respond in a different process, which then would write to either a .csv file or JSON file, and then in another process, I would then rank the responses so that the AI would not take so long to return something back. Also, I dislike how the API's cost money for effective use. If I want to build a program that relies on an AI such as GPT-5 or Gemini, that would mean that I have to pay for it, and even if I pay a lot of money, I still only get a limited amount of tokens per minute, and don't have an unlimited amount of responses. I do wish they would change that so people who do want to write solutions using AI may more freely do so.

I'm not sure if this was just because I wasn't passing something into the AI in the right way, but as I mentioned, it was very inconsistent in the amount of results it would give. Sometimes when I run it, it would give me 98 results, when the dataframe is 103 big. Then the next time I would run it, it would give me back 106 results when the dataframe was 103

big. I'm not sure if this was a problem with the code I was running, or if it was a problem with the Gemini API I was using. Either way, it kept giving inconsistent results, which ended up affecting my program overall.

Here are results from executing my python script:

Original:

| | query_id | precision@3 | recall@3 | nDCG@3 |
|---|---|---|---|---|
| 0 | 1 | 0.667 | 1.000 | 1.000 |
| 1 | 2 | 0.667 | 0.667 | 0.765 |
| 2 | 3 | 0.667 | 0.667 | 0.765 |
| 3 | 4 | 0.333 | 0.500 | 0.613 |
| 4 | 5 | 0.333 | 0.500 | 0.613 |
| 5 | 6 | 0.667 | 1.000 | 1.000 |
| 6 | 7 | 0.667 | 1.000 | 1.000 |
| 7 | 8 | 0.667 | 0.667 | 0.704 |
| 8 | 9 | 1.000 | 1.000 | 1.000 |
| 9 | 10 | 1.000 | 1.000 | 1.000 |
| 10 | 11 | 0.667 | 1.000 | 1.000 |
| 11 | 12 | 0.667 | 1.000 | 0.920 |
| 12 | 13 | 0.667 | 1.000 | 1.000 |
| 13 | 14 | 0.333 | 0.500 | 0.613 |
| 14 | 15 | 0.667 | 1.000 | 1.000 |
| 15 | 16 | 0.667 | 1.000 | 1.000 |
| 16 | 17 | 0.333 | 0.333 | 0.469 |
| 17 | 18 | 0.667 | 1.000 | 1.000 |

| | | | | |
|---|---|---|---|---|
| 18 | 19 | 0.667 | 1.000 | 1.000 |
| 19 | 20 | 0.667 | 1.000 | 1.000 |

AI:

Average metrics:

precision@3 0.633

recall@3 0.842

nDCG@3 0.873

dtype: float64

| | query_id | precision@3 | recall@3 | nDCG@3 |
|---|---|---|---|---|
| 0 | 1 | 0.333 | 0.500 | 0.613 |
| 1 | 2 | 0.333 | 0.333 | 0.235 |
| 2 | 3 | 0.333 | 0.333 | 0.469 |
| 3 | 4 | 0.667 | 1.000 | 0.693 |
| 4 | 5 | 0.667 | 1.000 | 1.000 |
| 5 | 6 | 0.667 | 1.000 | 0.920 |
| 6 | 7 | 0.667 | 1.000 | 0.920 |
| 7 | 8 | 0.667 | 0.667 | 0.765 |
| 8 | 9 | 0.667 | 0.667 | 0.704 |
| 9 | 10 | 0.667 | 0.667 | 0.765 |
| 10 | 11 | 0.667 | 1.000 | 0.920 |
| 11 | 12 | 0.333 | 0.500 | 0.613 |
| 12 | 13 | 0.667 | 1.000 | 0.920 |
| 13 | 14 | 0.667 | 1.000 | 0.920 |
| 14 | 15 | 0.667 | 1.000 | 0.693 |

| | | | | |
|---|---|---|---|---|
| 15 | 16 | 0.667 | 1.000 | 0.693 |
| 16 | 17 | 0.667 | 0.667 | 0.765 |
| 17 | 18 | 0.667 | 1.000 | 0.920 |
| 18 | 19 | 0.667 | 1.000 | 1.000 |
| 19 | 20 | 0.667 | 1.000 | 1.000 |

Average metrics:

precision@3 0.600

recall@3 0.817

nDCG@3 0.776

dtype: float64

Overall, good starting assignment, but I would like to figure out how to use the API for effectively in the future.