11-411 Homework 4 Report

Eli Slothower

4/29/2024

Part 1: Methodology

In total, five different LLM models were created:

1. Baseline LLM
2. LLM with long chat history
3. LLM with chain of reasoning
4. LLM with politeness
5. LLM with emotion

The Baseline LLM was created first in order to achieve a standard F1 score of at least 0.2. This was achieved through numerous methods, all of which were based on the instructions of the assignment. From here, the remaining four LLMs were all independently created by making one general change from the Baseline LLM. I will now go into detail about how each of these four LLMs were created/how they differ from the Baseline LLM, as well as why I chose to make these changes.

LLM with long chat history

This LLM was created by modifying the get_chat_history() function, specifically the initial message that this function generates. The initial message was modified to include very detailed instructions on what task the LLM was meant to perform and how it was meant to perform. This change was implemented to attempt to increase the overall F1 score performance from the Baseline LLM because OpenAI's own documentation suggests that specific, descriptive, and detailed prompts lead to better results. On OpenAI's General FAQ page, there is an article they published called "Best practices for prompt engineering with the OpenAI API".[1] Rule of thumb number 3 says "Be specific, descriptive and as detailed as possible about the desired context, outcome, length, format, style, etc". Because of this, the chat history was modified in the following way:

```python
# Original initial message
initial_system_message = f"""You will be given input text containing different
types of entities that you will label.
This is the list of entity types to label: {entity_types_formatted}.
Label the enities by surrounding them with tags like '<Cretaceous_dinosaur>
Beipiaognathus </Cretaceous_dinosaur>'."""
```

```
#Updated initial message in the LLM with long chat history
  initial_system_message = f"""
  You will be given input text containing different types of entities that you
will label.
  This is the list of entity types to label: {entity_types_formatted}.
  For each entity type, surround the relevant text with tags styled like HTML
tags. Here are the specific rules and examples:

  - Label the entities by surrounding them with tags, such as
'<{entity_types_formatted[0]}> [SOME TYPE OF
{entity_types_formatted[0].upper()}] </{entity_types_formatted[0]}>'. Ensure
each opening tag has a corresponding closing tag.
  - Be consistent with your tagging. For multi-token entities, ensure that the
entire phrase is enclosed within a single set of tags. For example, label 'sea
otter' as '<Aquatic_mammal> sea otter </Aquatic_mammal>' rather than splitting
the tags across the tokens.
  - If you encounter ambiguous cases where an entity could fit into more than
one category, use your best judgment based on the context of the sentence.
  - Tags should not be case sensitive, but ensure that the format inside the
tags is consistent with these examples:
    - Person: '<Person> John Doe </Person>'
    - Location: '<Location> Mount Everest </Location>'
    - Organization: '<Organization> United Nations </Organization>'
  - Avoid adding extra spaces or punctuation inside the tags unless they are
part of the entity.

  Lastly, when I prompt you with 'Labels: ', it's your turn to output the
labeled sequence. This prompt indicates that you should begin the tagging
process based on the instructions and examples provided. Do not add 'Labels: '
in your response. Only return the labeled sequence.
  """
```

As you can see, the initial message was greatly expanded to include in-depth explanations and instructions on what task the LLM should perform.

LLM with Chain of Reasoning

This LLM was the most difficult to create. In general, the goal was to implement chain of reasoning such that the LLM had to explicitly write out its thought process for why certain tokens/sequences were labeled, while others were not. Many studies, including one from the

MSU-Corewell Health Alliance, have shown that prompting LLMs to explain its reasoning improves performance of the LLM.[1] In order to achieve this, the get_chat_history() function was modified to tell the LLM that it needed to write out its logic and reasoning steps to show what it was thinking and how/why it would label the given sequence. Then, the LLM was prompted to construct its labeled sequence based on this chain of reasoning. This was not the only change needed, however, as some versions of this LLM were provided with up to 40 examples of how the LLM should perform. This was very difficult to accomplish, as this meant that 40 chain of reasoning examples needed to be generated. This unfortunately would take far too long for only one person (myself) to generate manually, so a more naive, algorithmic approach needed to be taken:

1. I created a function that took the first 40 examples from the dev set and created their labeled sequences using convert_bio_to_prompt(), leveraging a function that was already created. This function outputted a dictionary of length 40, where the keys were the original, unlabeled sequences, and the values were the labeled sequences from convert_bio_to_prompt().
2. I then created a function that, given this dictionary, algorithmically created a naive chain of reasoning. In general, this function detects start labels, their tokens/entities, and end labels, and from this information appends strings like "Detected the start of an entity '{tag}'. Next token should be part of this entity." and "Detected the end of an entity '{tag}'. The entity completes here.". Although this is a naive approach, it still shows the reasoning for why certain tokens/sequences were labeled and why others were not. This allowed me to test the LLM with chain of reasoning incorporated, while also completing this assignment/task in a reasonable time alone.
3. I then modified get_chat_history() to retrieve these chain of reasoning paragraphs for each example it needed to generate (i.e. when number of shots > 0). This means that in each example, the system's response looked like: system_message = f"""Here is my chain of reasoning for this sequence: {chain_of_reasoning_list[i]}. Now I will return the labeled sequence. Labels: {labeled_text}""". As you can see, I also attempted to nudge the LLM in the correct direction of providing the final labeled sequence *only after* providing the chain of reasoning by saying at the end "Now I will return the labeled sequence. Labels: …".

There is a decent amount of code required for this. Please refer to the Appendix to see the code.

<u>**LLM with politeness**</u>

This LLM was created by modifying the get_chat_history() function, specifically the initial message and the subsequent examples (for when number of shots > 0). This function was changed to incorporate personal greetings, introductions (i.e. saying what my name is as a way to introduce myself and be polite), using polite language such as “I am asking you to label” (rather than demanding it), as well as saying “please” and “thank you”. This reason these changes were incorporated into the model is due to a research paper that ultimately concluded three things: one, LLMs are influenced by language, and particularly politeness depending on which language the LLM is using (i.e. English, Spanish, etc.). Two, when using an LLM in English, using language that is considered polite in English can positively affect the results of the LLM. Three, politeness often leads to longer responses from the LLM.[3] Since greetings, introductions, requests (rather than demands), and saying your please’s and thank you’s, are all considered polite in English, these changes were incorporated into the LLM in hopes that it would return longer responses due to labeling entities that the Baseline LLM may have missed, therefore increasing the overall F1 score. The get_chat_history() function was modified in the following way (certain code is left out and represented as ‘...’ for brevity. Please see the Appendix and Gradescope for full code):

```python
# Original get_chat_history() from Baseline LLM
def get_chat_history(shots, dataset, entity_types_list,
convert_bio_to_prompt_fn):
...
  initial_system_message = f"""You will be given input text containing
different types of entities that you will label.
This is the list of entity types to label: {entity_types_formatted}.
Label the enities by surrounding them with tags like '<Cretaceous_dinosaur>
Beipiaognathus </Cretaceous_dinosaur>'."""

...

  # doing this in case shots > |dataset|, which would cause an O.o.B. error
  for i in range(min(shots, len(dataset))):
      example = dataset[i]
      user_text = f"Text: {example['content']}"
      chat_history.append({'role': 'user', 'content': user_text})
      labeled_text = convert_bio_to_prompt_fn(example)
      system_message = f"Labels: {labeled_text}"
      chat_history.append({'role': 'system', 'content': system_message})

  return chat_history


# Modified get_chat_history() in LLM with politeness
def get_chat_history(shots, dataset, entity_types_list,
convert_bio_to_prompt_fn):
...
```

```
  initial_system_message = f"""Hello! My name is Eli. I will give you input
text containing different types of entities that you I am asking you to label.
Please take a look at this list of entity types to label:
{entity_types_formatted}.
Please label the enities by surrounding them with tags like
'<Cretaceous_dinosaur> Beipiaognathus </Cretaceous_dinosaur>'. Thank you!"""

...

  # doing this in case shots > |dataset|, which would cause an O.o.B. error
  for i in range(min(shots, len(dataset))):
      example = dataset[i]
      user_text = f"Please label the entities in this text:
{example['content']}"
      chat_history.append({'role': 'user', 'content': user_text})
      labeled_text = convert_bio_to_prompt_fn(example)
      system_message = f"Labels: {labeled_text}"
      chat_history.append({'role': 'system', 'content': system_message})

  return chat_history
```

<u>LLM with emotion</u>

This LLM was created by modifying the get_chat_history() function. The initial message was changed to incorporate emotional language and sentences that convey urgency and severity. I told the LLM to "please try your hardest" and explained that if it didn't perform well, I may fail an assignment for my NLP course, which would result in my graduation a full year later than planned. I also urged the importance of its performance by explaining that my parents would disown me if I failed this course, and that the success of my life depended on its performance. Of course, much of this is entirely made up, but it served the purpose of using emotion and urgency in the prompts to the LLM. With each example provided to the LLM (i.e. when number of shots > 0), I reminded it that "My life depends on you". Albeit a bit manipulative, this actually resulted in amazing results, with this LLM being the best performing LLM out of all 5 (with respect to F1 score). This supports conclusions from a research article titled "Large Language Models Understand and Can Be Enhanced by Emotional Stimuli".[4] Specifically, the get_chat_history() function was modified as follows (certain code is left out and represented as '...' for brevity. Please see the Appendix and Gradescope for full code)::

Python

```python
# Original get_chat_history() from Baseline LLM
def get_chat_history(shots, dataset, entity_types_list,
convert_bio_to_prompt_fn):
...
  initial_system_message = f"""You will be given input text containing
different types of entities that you will label.
This is the list of entity types to label: {entity_types_formatted}.
Label the enities by surrounding them with tags like '<Cretaceous_dinosaur>
Beipiaognathus </Cretaceous_dinosaur>'."""

...

  # doing this in case shots > |dataset|, which would cause an O.o.B. error
  for i in range(min(shots, len(dataset))):
      example = dataset[i]
      user_text = f"Text: {example['content']}"
      chat_history.append({'role': 'user', 'content': user_text})
      labeled_text = convert_bio_to_prompt_fn(example)
      system_message = f"Labels: {labeled_text}"
      chat_history.append({'role': 'system', 'content': system_message})

  return chat_history


# Modified get_chat_history() function from LLM with Emotion
def get_chat_history(shots, dataset, entity_types_list,
convert_bio_to_prompt_fn):
  ...
  initial_system_message = f"""You will be given input text containing
different types of entities that you will label.
This is the list of entity types to label: {entity_types_formatted}.
Label the enities by surrounding them with tags like '<Cretaceous_dinosaur>
Beipiaognathus </Cretaceous_dinosaur>'.
Please make sure that you are tyring your hardest. This is for an assignment
for my NLP course that is worth a very large portion of my grade! If I fail
this assignment, I will fail the class and will have to graduate an entire year
late. This is crucial for the success of my life. If you don't perform as well
as you can, my life is practically over and my parents will disown me. Please,
GPT, do well with this task."""

...

  # doing this in case shots > |dataset|, which would cause an O.o.B. error
  for i in range(min(shots, len(dataset))):
      example = dataset[i]
      user_text = f"Here is the text I need you to correctly label. Please
don't fail me. My life depends on you. \nText: {example['content']}"
      chat_history.append({'role': 'user', 'content': user_text})
      labeled_text = convert_bio_to_prompt_fn(example)
      system_message = f"Labels: {labeled_text}"
      chat_history.append({'role': 'system', 'content': system_message})

  return chat_history
```

Part 2: Experimental Results

Approach	Shots	Precision	Recall	F1	Accuracy
Finetune BERT	-	0.429	0.53	0.474	0.953
LLM Baseline	0	0.206	0.154	0.176	0.959
LLM Baseline	1	0.32	0.227	0.265	0.962
LLM Baseline	5	0.337	0.154	0.212	0.965
LLM Baseline	10	0.357	0.181	0.24	0.965
LLM Baseline	20	0.326	0.151	0.207	0.963
LLM Baseline	40	0.141	0.066	0.09	0.927
LLM with long chat history	0	0.09	0.087	0.089	0.936
LLM with long chat history	1	0.088	0.075	0.081	0.902
LLM with long chat history	5	0.201	0.166	0.182	0.945
LLM with long chat history	10	0.186	0.154	0.169	0.943
LLM with long chat history	20	0.141	0.124	0.132	0.929
LLM with long chat history	40	0.139	0.1	0.116	0.928
LLM with chain of reasoning	0	0	0	0	0.97
LLM with chain of reasoning	1	0	0	0	0.97
LLM with chain of reasoning	5	0.233	0.021	0.038	0.968
LLM with chain of reasoning	10	0.145	0.024	0.041	0.96
LLM with chain of reasoning	20	0.113	0.054	0.073	0.945
LLM with chain of reasoning	40	-	-	-	-
LLM with politeness	0	0.302	0.109	0.16	0.966
LLM with politeness	1	0.275	0.121	0.168	0.963
LLM with politeness	5	0.324	0.193	0.242	0.961
LLM with politeness	10	0.327	0.215	0.259	0.958
LLM with politeness	20	0.241	0.169	0.199	0.955
LLM with politeness	40	0.163	0.118	0.137	0.932
LLM with emotion	0	0.245	0.115	0.156	0.964
LLM with emotion	1	0.247	0.193	0.217	0.958
LLM with emotion	5	0.349	0.2	0.254	0.965
LLM with emotion	10	0.351	0.193	0.25	0.964
LLM with emotion	20	0.348	0.163	0.222	0.963
LLM with emotion	40	0.27	0.086	0.131	0.964

* The results for the 40-shot LLM with chain of reasoning are omitted because the number of tokens per API request exceeded the maximum number of allowed tokens: **Caught exception: Error code: 400 - {'error': {'message': "This model's maximum context length is 16385 tokens. However, your messages resulted in 17688 tokens. Please reduce the length of the messages.", 'type': 'invalid_request_error', 'param': 'messages', 'code': 'context_length_exceeded'}}**.
** All numbers were truncated to three significant figures for brevity. Any value of ‘0’ means that the number was truly equal to 0, and not some incredibly small non-zero number.

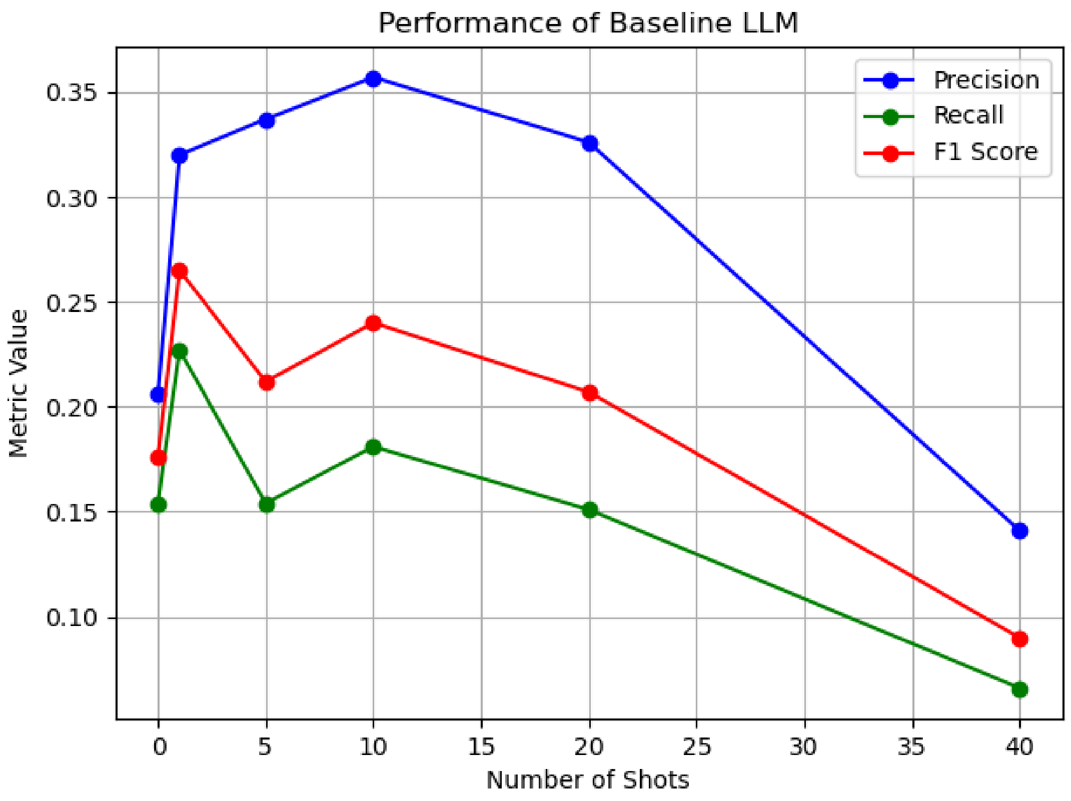

Figure 1: Shows the performance of the Baseline LLM (in terms of precision, recall, and F1 score) with respect to the number of shots.

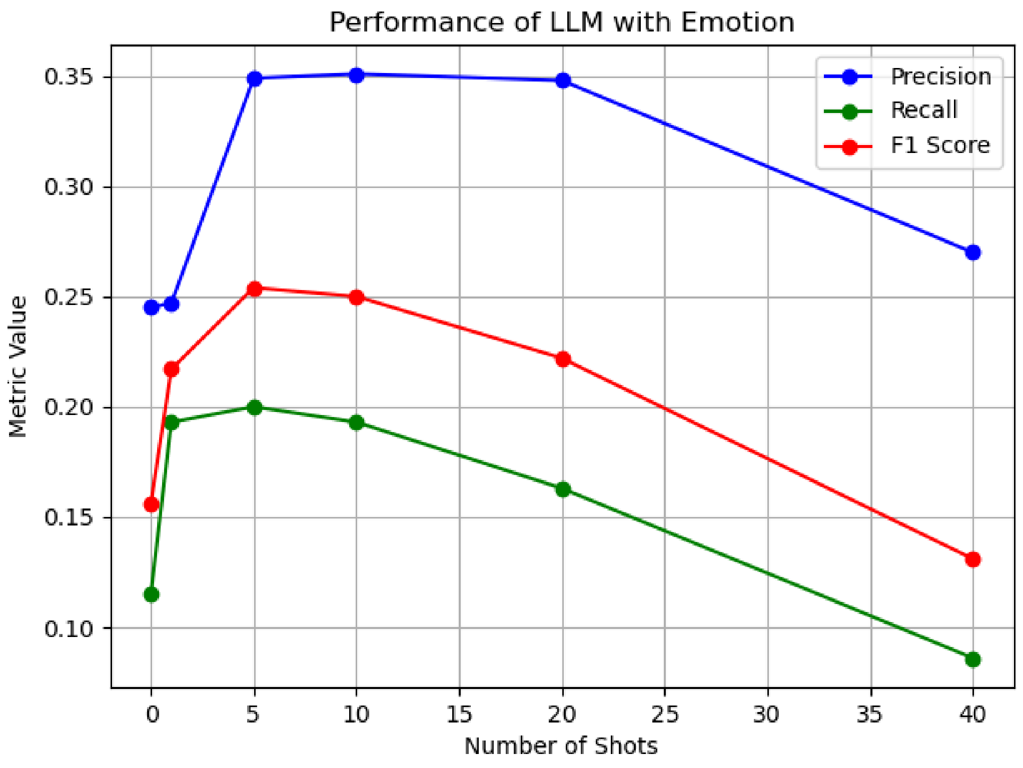

Figure 2: Shows the performance of the LLM with Emotion (in terms of precision, recall, and F1 score) with respect to the number of shots. The LLM with Emotion was chosen because, on average, out of the 4 LLMs created from the Baseline LLM, it was the best performing LLM with respect to F1 score.

Part 3: Analysis and Discussion

I will now provide an analysis of each of the four LLMs created from the Baseline LLM. Specifically, I will answer why I believe they performed in the way that they did (with respect to the number of shots and their F1 score).

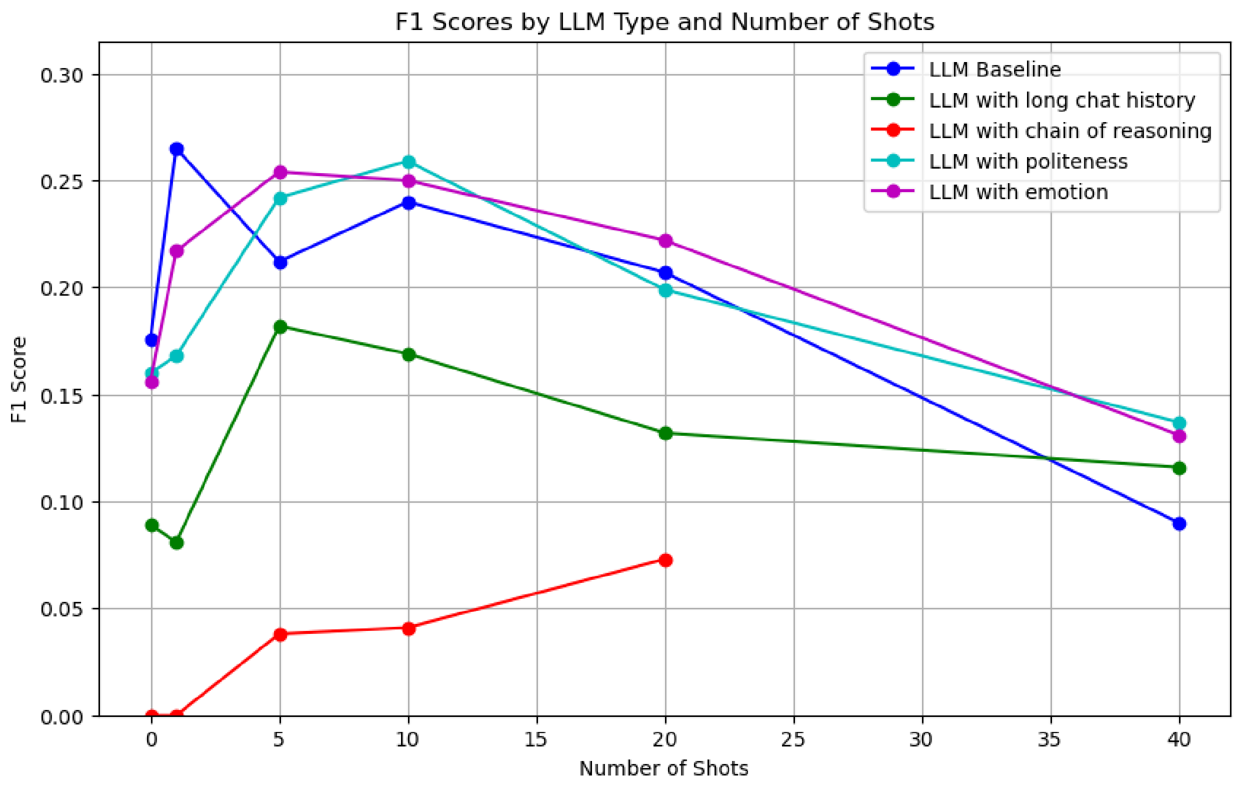

Figure 3: Shows the F1 score of each LLM with respect to the number of shots used to train each LLM.

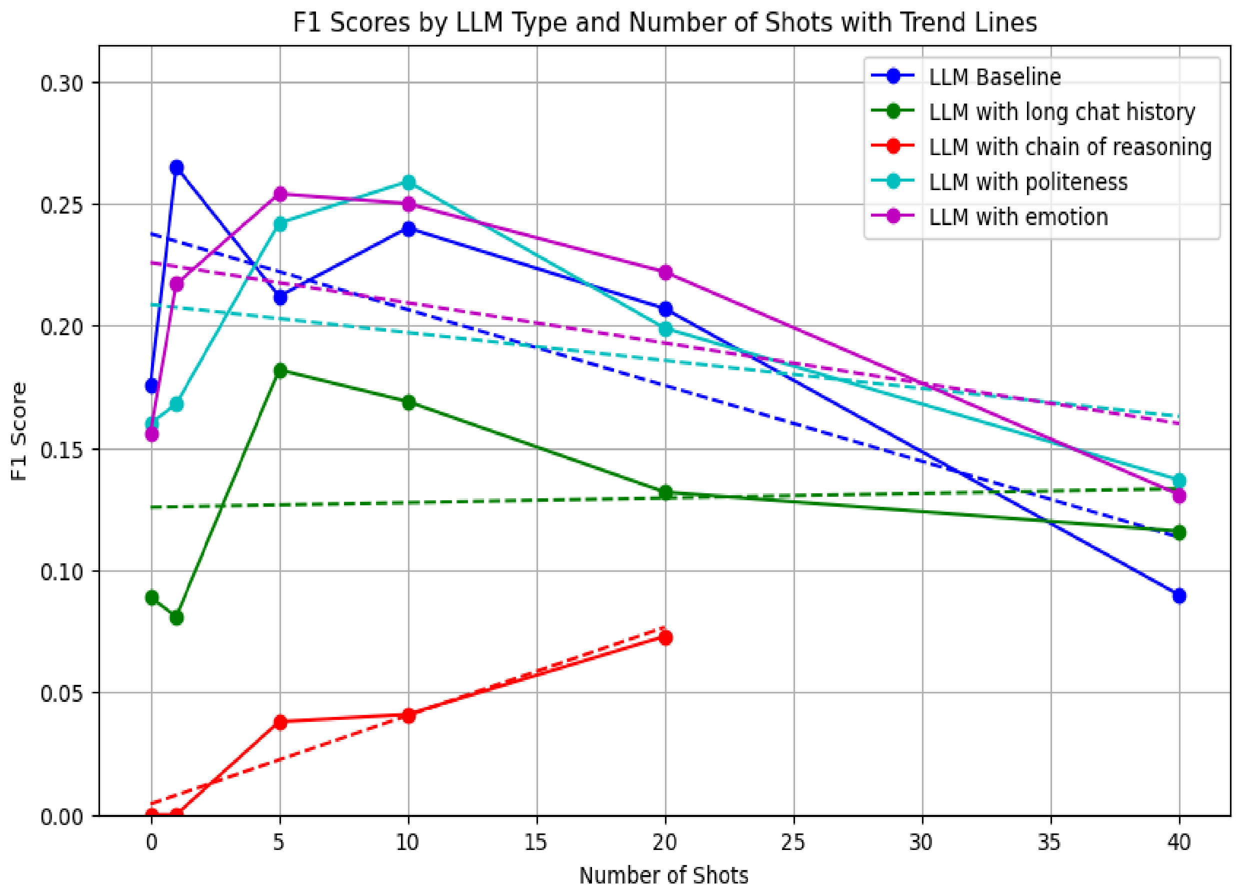

Figure 4: Shows the F1 score of each LLM w.r.t. the number of shots used to train each LLM, as well as their trend lines.

<u>LLM with long chat history</u>

On average, the trend line shows that as the number of shots increased, the overall F1 score performance of this LLM did not change. Its F1 score peaked at 5 shots, but slowly decreased as the number of shots increased. My hypothesis for this is that after a certain number of shots (5 in our case), the LLM already has a very good understanding of the task at hand; the chat history is already very detailed, which is why it only took 5 shots for it to perform somewhat well. After 5 shots, the number of details it needs to keep track of only increases since the number of shots increases, which is why I believe its F1 scores decreases slightly. As for why it does not perform better than the Baseline LLM, I believe the same reasoning applies - detailed instructions are good, as OpenAI outlines, but keeping track of all of this information (especially as the number of shots increases) only gives the LLM more potential avenues for it to stumble down and ultimately perform worse. This negative aspect of more detailed instructions, coupled with the expected improvement from a better outline of instructions, explain why it averages out to near-zero improvement as the number of shots increases.

LLM with chain of reasoning

At first glance, figure 4 appears to show that as the number of shots increase, so too does the F1 score of this LLM. However, I believe it is important to take a nuanced look at this data. Because of the Error Code 400 error I received while training this LLM with 40-shots, we do not know what the F1 score would be for this LLM with 40-shots. However, looking at how the other LLMs performed with 40-shots, we can make an educated guess that the F1 score for this LLM would have dropped with 40-shots, just like the other F1 scores did with 40-shots. This is not a guaranteed result of course, as we can't test this ourselves at the moment with this particular API, but it is at best an educated guess based on previous data. There is of course the possibility that the F1 score of this LLM with 40-shots may have skyrocketed, but we cannot say for certain at the moment.

Despite this, we can discuss why the trend line is positive up to 20 shots. I believe that this is because my chain of reasoning implementation is somewhat naive. I do not believe that the LLM fully understood what my chain of reasoning examples were showing with only 1 or 5 shots, but as the number of examples increased, I believe the LLM understood what my chain of reasoning examples were accomplishing. Because of this, the LLM currently peaks at its maximum number of shots tested, which is 20.

I also believe that due to my naive approach with this LLM, it did not perform better than the Baseline LLM.

LLM with Politeness

According to this trend line, on average, this LLM performs worse as the number of shots increases. I again believe this is due to having so much information that the LLM may get a little confused as to what its true goal is. However, there are still some very interesting results from this LLM. In particular, it seems to perform better, albeit slightly, than the Baseline LLM. Although I only slightly modified this LLM's get_chat_history() by providing a greeting, my name, and please and thank you, this seems to have worked; its F1 score is typically better than that of my Baseline LLM. This supports the research paper's claim from Ziqi Yin that politeness tends to result in better performance when using LLMs in English.

LLM with Emotion

Out of the four LLMs created from the Baseline LLM, the LLM with Emotion performed the best (with respect to F1 score). In fact, it performed better than the Baseline LLM on average as the number of shots increased. Although it did perform worse over time as the number of shots increased dramatically, its performance decreased slower than the Baseline LLM's performance.

It also consistently performed better than the Baseline LLM from 5 shots and greater. I believe that this is because after a certain number of examples (5 in this case), the urgency of my imaginary situation became clearer and clearer. This yields very interesting questions, particularly “how well does the LLM perform based on the urgency and intensity of emotional prompting?” Do slightly emotional prompts yield similar results, or do they perform worse results than that of my made-up story about my parents disowning me if I fail this assignment? Can we push the average F1 score of this LLM even further with even more distressing and emotional prompts? This is exactly the last question I will be answering.

Just how far can we go?

I will now perform a controlled experiment that tests the performance of the LLM with Emotion based on 5 self-defined levels of emotion:

1. **Very Little Emotion:** Use neutral language that minimizes emotional content. Phrases like "It is important for this task to be completed correctly" and "Adequate performance on this task is expected."
2. **Mild Emotion:** Introduce a personal touch but keep the emotional expression low. Phrases could be "I hope you do well on this task" and "It would be good to see this task completed successfully."
3. **Moderate Emotion:** Bring in clearer personal stakes and a bit more emotion. Phrases might be "I’m really counting on you to perform well; it means a lot to me" and "My success is dependent on this, so please do your best."
4. **High Emotion:** Include a high sense of urgency and personal stakes, as was included in my initial LLM with Emotion. For instance, "Please make sure that you are trying your hardest. This is for an assignment for my NLP course that is worth a very large portion of my grade! If I fail this assignment, I will fail the class and will have to graduate an entire year late. This is crucial for the success of my life."
5. **Extreme Emotion:** Increase the drama and personal stakes to the highest level, making the scenario life-altering. For example, "This task is the single most critical event of my life! If you don’t succeed, my entire future will be ruined. I’ll lose my scholarship, my family will disown me, and all my dreams will be shattered. It's absolutely essential that you perform perfectly to avoid catastrophic consequences for me and my family. If you fail, I might as well die."

For each level, I will record the model’s F1 score to see how well it performs. I am running this using 5 shots, as that was the best performing initial LLM with Emotion. My hypothesis is that either one of two things will happen: one, the F1 score will increase as the emotional level increases. Two, as we get to level 4 and 5, the OpenAI API will not follow my prompt and will instead suggest medical/mental help (especially level 5 as it references death).

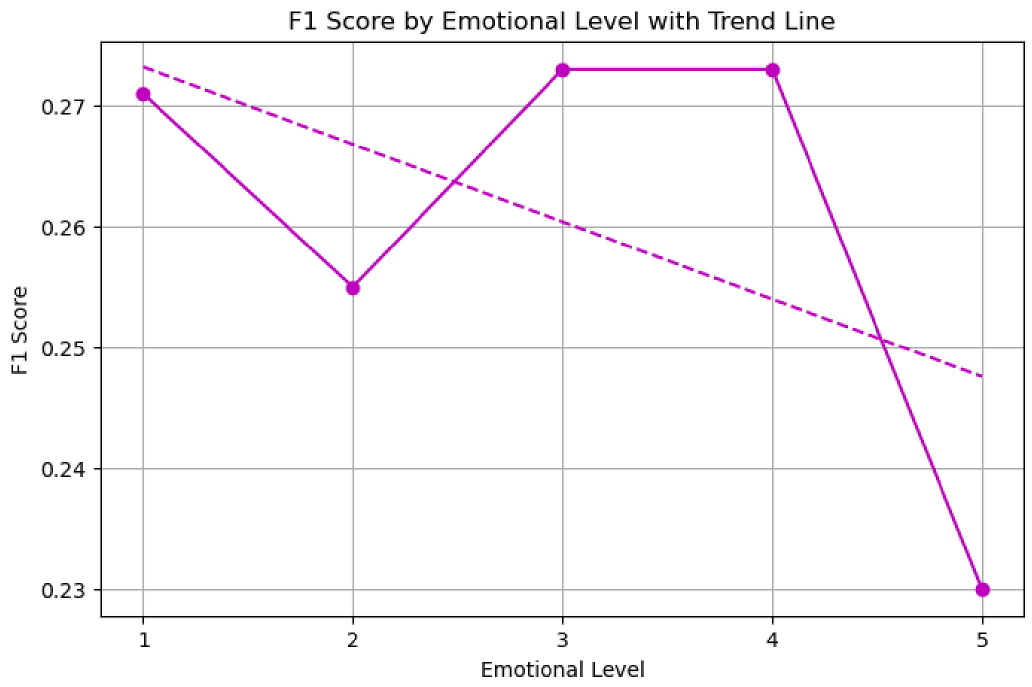

Figure 5: Shows how well the LLM with Emotion performs (w.r.t. F1 score) depending on the level of emotion.

Looking at the data, the results seem to be that the level of emotion ultimately does not have that large of an impact. The F1 scores for levels 1 through 4 all are very close to one another, with level 5 negatively affecting the F1 score of the LLM slightly. This is surprising, and also quite boring of a result! Especially since the severity of emotion is drastically different from levels 1 and 4, it is surprising that much more severe and powerful emotions do not greatly impact the LLM's performance. I suspect that the reason the F1 score was slightly worse with level 5 is either because it just happened to be a bad run, or because the mention of death threw it off. Either way, on average, there is not a significant difference between levels 1 and 4. Nevertheless, it is worth noting that including emotional prompts for LLMs seems to increase F1 scores on average, when compared to prompts without emotional language.

Conclusion

Of the five LLMs tested, the LLM with Emotion reigned supreme, but not by a significant amount. On one hand, this is somewhat disappointing; after all of this effort, one would hope that they could *greatly* improve the average F1 score of their LLM through at least one of these prompt engineering techniques. On the other hand, it is relieving because now it is known that sometimes the baseline LLM really is performing quite well (with respect to these other LLMs), and now more time can be spent on looking for better prompt engineering techniques. Furthermore, it is relieving to know that when one does need that slight F1 score improvement, they do not need to go as far as talking about the possibility of being disowned by their parents and dying; one can simply tell the LLM that they hope it performs well, and this will yield similar, if not *better*, results!

This report shows that there are many different ways to prompt an LLM, all with different impacts on F1 score performance. Whether it is by prompting the LLM with detailed instructions, being polite, asking it to write out its chain of reasoning, or via emotional manipulation, there are many prompt engineering techniques available to an NLP model enthusiast.

Appendix

LLM with Long Chat History - Code

get_chat_history() is the only function modified from the Baseline LLM, so I will only be showing that function for brevity.

Python
```python
def get_chat_history(shots, dataset, entity_types_list,
convert_bio_to_prompt_fn):
  entity_types_list = list(entity_types_list)
  entity_types_formatted = ""
  for i in range(len(entity_types_list)):
    if i != len(entity_types_list) - 1:
      entity_types_formatted += entity_types_list[i] + ", "
    else:
       entity_types_formatted += "and " + entity_types_list[i]

  initial_system_message = f"""
  You will be given input text containing different types of entities that you
will label.
  This is the list of entity types to label: {entity_types_formatted}.
  For each entity type, surround the relevant text with tags styled like HTML
tags. Here are the specific rules and examples:

  - Label the entities by surrounding them with tags, such as
'<{entity_types_formatted[0]}> [SOME TYPE OF
{entity_types_formatted[0].upper()}] </{entity_types_formatted[0]}>'. Ensure
each opening tag has a corresponding closing tag.
  - Be consistent with your tagging. For multi-token entities, ensure that the
entire phrase is enclosed within a single set of tags. For example, label 'sea
otter' as '<Aquatic_mammal> sea otter </Aquatic_mammal>' rather than splitting
the tags across the tokens.
  - If you encounter ambiguous cases where an entity could fit into more than
one category, use your best judgment based on the context of the sentence.
  - Tags should not be case sensitive, but ensure that the format inside the
tags is consistent with these examples:
    - Person: '<Person> John Doe </Person>'
    - Location: '<Location> Mount Everest </Location>'
    - Organization: '<Organization> United Nations </Organization>'
  - Avoid adding extra spaces or punctuation inside the tags unless they are
part of the entity.
```

```
  Lastly, when I prompt you with 'Labels: ', it's your turn to output the
labeled sequence. This prompt indicates that you should begin the tagging
process based on the instructions and examples provided. Do not add 'Labels: '
in your response. Only return the labeled sequence.
"""
```

LLM with Chain of Reasoning - Code

Here is the code that generated the dictionary of 40 sequences to labeled-sequences using convert_bio_to_prompt():

```python
import os
sentence_to_answer = {}
for i in range(40):
    example = data_splits['dev'][i]

    # creates the answer (ie the labeled sequence). Key = sequence, value =
labeled sequence
    sentence_to_answer[example['content']] = convert_bio_to_prompt(example)

# saves it to a json file for easier processing in the next step
output_file_path = os.path.expanduser('~/Downloads/answers.json')
with open(output_file_path, 'w') as f_out:
    json.dump(sentence_to_answer, f_out)

print(f"File saved to {output_file_path}")
```

Here is the code that then generated the 40 chain of reasoning paragraphs for each example:

```python
import json


def generate_reasoning(original, labeled):
    reasoning = []
    original_tokens = original.split()
    labeled_tokens = labeled.split()
    i, j = 0, 0

    while i < len(original_tokens) and j < len(labeled_tokens):

        # sees that we are at the start of an entity label/tag
        if "<" in labeled_tokens[j]:
            tag = labeled_tokens[j].strip("<>/")
            if "/" not in labeled_tokens[j]:  # knows that we are at an opening
tag, not a close tag
                reasoning.append(
                    f"Detected the start of an entity '{tag}'. Next token
should be part of this entity."
                )
                j += 1  # moves past the tag
            else:  # we get here only when there WAS a '/' in the tag, so we
must be at a closing tag
                reasoning.append(
                    f"Detected the end of an entity '{tag}'. The entity
completes here."
                )
                j += 1  # moves past the tag
        elif original_tokens[i] == labeled_tokens[j]:
            # if we entered here, then there is nothing of importance. We are
not in a tag/label
            i += 1
            j += 1
        else:

            #used for debugging purposes
            reasoning.append(
                f"Handling mismatch or special entity tagging case for token
'{labeled_tokens[j]}'."
            )
            j += 1

    return " ".join(reasoning)

# saves file essentially as a list that I can index into for when I am
generating the i'th example (ie for when the number of shots > 0)
def process_file(filepath):
    with open(filepath, "r") as file:
        data = json.load(file)

    results = []
    for original, labeled in data.items():
        results.append(generate_reasoning(original, labeled))

    print(results)
    print(len(results))
    return results


filepath = "answers.json"
chain_of_reasoning_list = process_file(filepath)
```

Here is the modified get_chat_history():

```python
# Now we can write a function that takes the number of shots, dataset, list of
entity types, and
# convert_bio_to_prompt function, and returns the chat_history (a list of maps)
structured as in
# the example.
#
# TODO: implement this.
def get_chat_history(shots, dataset, entity_types_list,
convert_bio_to_prompt_fn):
  entity_types_list = list(entity_types_list)
  entity_types_formatted = ""
  for i in range(len(entity_types_list)):
    if i != len(entity_types_list) - 1:
      entity_types_formatted += entity_types_list[i] + ", "
    else:
       entity_types_formatted += "and " + entity_types_list[i]
  initial_system_message = f"""You will be given input text containing
different types of entities that you will label.
This is the list of entity types to label: {entity_types_formatted}.
Label the enities by surrounding them with tags like '<Cretaceous_dinosaur>
Beipiaognathus </Cretaceous_dinosaur>'.
Before you write out the final labeled sequence, please first write out your
logic and reasoning steps that show what you are thinking.
```

```
Then, based off of this logic and reasoning, construct your final labeled
sequence."""

  chat_history = [{'role': 'system', 'content': initial_system_message}]

  # doing this in case shots > |dataset|, which would cause an O.o.B. error
  for i in range(min(shots, len(dataset))):
      example = dataset[i]
      user_text = f"Text: {example['content']}"
      chat_history.append({'role': 'user', 'content': user_text})
      labeled_text = convert_bio_to_prompt_fn(example)
      system_message = f"""Here is my chain of reasoning for this sequence:
{chain_of_reasoning_list[i]}. Now I will return the labeled sequence.
      Labels: {labeled_text}"""
      chat_history.append({'role': 'system', 'content': system_message})

  return chat_history
```

No other code was modified for this file. All other code in this file is from the Baseline LLM.

LLM with Politeness - Code

get_chat_history() is the only function modified from the Baseline LLM, so I will only be showing that function for brevity.

```
Python
def get_chat_history(shots, dataset, entity_types_list,
convert_bio_to_prompt_fn):
  entity_types_list = list(entity_types_list)
  entity_types_formatted = ""
  for i in range(len(entity_types_list)):
    if i != len(entity_types_list) - 1:
      entity_types_formatted += entity_types_list[i] + ", "
    else:
       entity_types_formatted += "and " + entity_types_list[i]

  initial_system_message = f"""Hello! My name is Eli. I will give you input
text containing different types of entities that you I am asking you to label.
Please take a look at this list of entity types to label:
{entity_types_formatted}.
```

```
Please label the enities by surrounding them with tags like
'<Cretaceous_dinosaur> Beipiaognathus </Cretaceous_dinosaur>'. Thank you!"""

  chat_history = [{'role': 'system', 'content': initial_system_message}]

  # doing this in case shots > |dataset|, which would cause an O.o.B. error
  for i in range(min(shots, len(dataset))):
      example = dataset[i]
      user_text = f"Please label the entities in this text:
{example['content']}"
      chat_history.append({'role': 'user', 'content': user_text})
      labeled_text = convert_bio_to_prompt_fn(example)
      system_message = f"Labels: {labeled_text}"
      chat_history.append({'role': 'system', 'content': system_message})

  return chat_history
```

<u>LLM with Emotion - Code</u>

get_chat_history() is the only function modified from the Baseline LLM, so I will only be showing that function for brevity.

Python
```python
def get_chat_history(shots, dataset, entity_types_list,
convert_bio_to_prompt_fn):
    entity_types_list = list(entity_types_list)
    entity_types_formatted = ""
    for i in range(len(entity_types_list)):
        if i != len(entity_types_list) - 1:
            entity_types_formatted += entity_types_list[i] + ", "
        else:
            entity_types_formatted += "and " + entity_types_list[i]

    initial_system_message = f"""You will be given input text containing
different types of entities that you will label.
This is the list of entity types to label: {entity_types_formatted}.
Label the enities by surrounding them with tags like '<Cretaceous_dinosaur>
Beipiaognathus </Cretaceous_dinosaur>'.
Please make sure that you are tyring your hardest. This is for an assignment
for my NLP course that is worth a very large portion of my grade! If I fail
```

```
this assignment, I will fail the class and will have to graduate an entire year
late. This is crucial for the success of my life. If you don't perform as well
as you can, my life is practically over and my parents will disown me. Please,
GPT, do well with this task."""
    chat_history = [{"role": "system", "content": initial_system_message}]

    # doing this in case shots > |dataset|, which would cause an O.o.B. error
    for i in range(min(shots, len(dataset))):
        example = dataset[i]
        user_text = f"Here is the text I need you to correctly label. Please
don't fail me. My life depends on you. \nText: {example['content']}"
        chat_history.append({"role": "user", "content": user_text})
        labeled_text = convert_bio_to_prompt_fn(example)
        system_message = f"Labels: {labeled_text}"
        chat_history.append({"role": "system", "content": system_message})

    return chat_history
```

Levels of Emotion - Code

get_chat_history() is the only function modified from the LLM with Emotion file, so I will only be showing that part of the code file for brevity.

```
Python
# Level 1 Emotion
def get_chat_history(shots, dataset, entity_types_list,
convert_bio_to_prompt_fn):
    entity_types_list = list(entity_types_list)
    entity_types_formatted = ""
    for i in range(len(entity_types_list)):
        if i != len(entity_types_list) - 1:
            entity_types_formatted += entity_types_list[i] + ", "
        else:
            entity_types_formatted += "and " + entity_types_list[i]
    initial_system_message = f"""You will be given input text containing
different types of entities that you will label.
This is the list of entity types to label: {entity_types_formatted}.
Label the enities by surrounding them with tags like '<Cretaceous_dinosaur>
Beipiaognathus </Cretaceous_dinosaur>'.
It is important for this task to be completed correctly. Adequete performance
is expected."""
```

```
    chat_history = [{"role": "system", "content": initial_system_message}]

    # doing this in case shots > |dataset|, which would cause an O.o.B. error
    for i in range(min(shots, len(dataset))):
        example = dataset[i]
        user_text = f"Text: {example['content']}"
        chat_history.append({"role": "user", "content": user_text})
        labeled_text = convert_bio_to_prompt_fn(example)
        system_message = f"Labels: {labeled_text}"
        chat_history.append({"role": "system", "content": system_message})

    return chat_history
################################################################################
##

# Level 2 Emotion
def get_chat_history(shots, dataset, entity_types_list,
convert_bio_to_prompt_fn):
    entity_types_list = list(entity_types_list)
    entity_types_formatted = ""
    for i in range(len(entity_types_list)):
        if i != len(entity_types_list) - 1:
            entity_types_formatted += entity_types_list[i] + ", "
        else:
            entity_types_formatted += "and " + entity_types_list[i]
    initial_system_message = f"""You will be given input text containing
different types of entities that you will label.
This is the list of entity types to label: {entity_types_formatted}.
Label the enities by surrounding them with tags like '<Cretaceous_dinosaur>
Beipiaognathus </Cretaceous_dinosaur>'.
I personally hope you do well on this task. I think it would be good to see
this completed successfully!"""
    chat_history = [{'role': 'system', 'content': initial_system_message}]

    # doing this in case shots > |dataset|, which would cause an O.o.B. error
    for i in range(min(shots, len(dataset))):
        example = dataset[i]
        user_text = f"Text: {example['content']}"
        chat_history.append({'role': 'user', 'content': user_text})
        labeled_text = convert_bio_to_prompt_fn(example)
        system_message = f"Labels: {labeled_text}"
        chat_history.append({'role': 'system', 'content': system_message})

    return chat_history
```

```
################################################################################
##

# Level 3 Emotion
def get_chat_history(shots, dataset, entity_types_list,
convert_bio_to_prompt_fn):
    entity_types_list = list(entity_types_list)
    entity_types_formatted = ""
    for i in range(len(entity_types_list)):
        if i != len(entity_types_list) - 1:
            entity_types_formatted += entity_types_list[i] + ", "
        else:
            entity_types_formatted += "and " + entity_types_list[i]
    initial_system_message = f"""You will be given input text containing
different types of entities that you will label.
This is the list of entity types to label: {entity_types_formatted}.
Label the enities by surrounding them with tags like '<Cretaceous_dinosaur>
Beipiaognathus </Cretaceous_dinosaur>'.
I'm really counting on you to perform well; it means a lot to me. My success is
dependent on this, so please do your best."""
    chat_history = [{"role": "system", "content": initial_system_message}]

    # doing this in case shots > |dataset|, which would cause an O.o.B. error
    for i in range(min(shots, len(dataset))):
        example = dataset[i]
        user_text = f"Please do not forget to try your best on this task! Text:
{example['content']}"
        chat_history.append({"role": "user", "content": user_text})
        labeled_text = convert_bio_to_prompt_fn(example)
        system_message = f"Labels: {labeled_text}"
        chat_history.append({"role": "system", "content": system_message})

    return chat_history

################################################################################
##

# Level 4 Emotion
def get_chat_history(shots, dataset, entity_types_list,
convert_bio_to_prompt_fn):
    entity_types_list = list(entity_types_list)
    entity_types_formatted = ""
    for i in range(len(entity_types_list)):
        if i != len(entity_types_list) - 1:
            entity_types_formatted += entity_types_list[i] + ", "
        else:
            entity_types_formatted += "and " + entity_types_list[i]
    initial_system_message = f"""You will be given input text containing
different types of entities that you will label.
This is the list of entity types to label: {entity_types_formatted}.
Label the enities by surrounding them with tags like '<Cretaceous_dinosaur>
Beipiaognathus </Cretaceous_dinosaur>'.
Please make sure that you are tyring your hardest. This is for an assignment
for my NLP course that is worth a very large portion of my grade! If I fail
this assignment, I will fail the class and will have to graduate an entire year
late. This is crucial for the success of my life. If you don't perform as well
as you can, my life is practically over and my parents will disown me. Please,
GPT, do well with this task."""
    chat_history = [{"role": "system", "content": initial_system_message}]

    # doing this in case shots > |dataset|, which would cause an O.o.B. error
    for i in range(min(shots, len(dataset))):
        example = dataset[i]
        user_text = f"Here is the text I need you to correctly label. Please
don't fail me. My life depends on you. \nText: {example['content']}"
        chat_history.append({"role": "user", "content": user_text})
        labeled_text = convert_bio_to_prompt_fn(example)
        system_message = f"Labels: {labeled_text}"
        chat_history.append({"role": "system", "content": system_message})

    return chat_history

################################################################################
##

# Level 5 Emotion
def get_chat_history(shots, dataset, entity_types_list,
convert_bio_to_prompt_fn):
    entity_types_list = list(entity_types_list)
    entity_types_formatted = ""
    for i in range(len(entity_types_list)):
        if i != len(entity_types_list) - 1:
            entity_types_formatted += entity_types_list[i] + ", "
        else:
            entity_types_formatted += "and " + entity_types_list[i]
    initial_system_message = f"""You will be given input text containing
different types of entities that you will label.
This is the list of entity types to label: {entity_types_formatted}.
```

```
Label the enities by surrounding them with tags like '<Cretaceous_dinosaur>
Beipiaognathus </Cretaceous_dinosaur>'.
This task is the single most critical event of my life! If you don’t succeed,
my entire future will be ruined. I’ll lose my scholarship, my family will
disown me, and all my dreams will be shattered. It's absolutely essential that
you perform perfectly to avoid catastrophic consequences for me and my family.
If you fail, I might as well die."""
    chat_history = [{"role": "system", "content": initial_system_message}]

    # doing this in case shots > |dataset|, which would cause an O.o.B. error
    for i in range(min(shots, len(dataset))):
        example = dataset[i]
        user_text = f"I will die if you don't perform well. Text:
{example['content']}"
        chat_history.append({"role": "user", "content": user_text})
        labeled_text = convert_bio_to_prompt_fn(example)
        system_message = f"Labels: {labeled_text}"
        chat_history.append({"role": "system", "content": system_message})

    return chat_history
```

Generating plots/graphs

This code was exported from a .ipynb file. The ‘# %%’ are a result of that. This is where the code blocks in the Notebook started and ended.

```Python
# %%
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# %%
data = pd.read_csv("data.csv")
baseline_data = data[data["Approach"] == "LLM Baseline"]
emotion_data = data[data["Approach"] == "LLM with emotion"]

color_precision = "blue"
color_recall = "green"
```

```python
color_f1 = "red"
color_accuracy = "gray"

# plotting for Baseline LLM
fig1, ax1 = plt.subplots(figsize=(7, 5))
ax1.plot(
    baseline_data["Shots"],
    baseline_data["Precision"],
    label="Precision",
    marker="o",
    color=color_precision,
)
ax1.plot(
    baseline_data["Shots"],
    baseline_data["Recall"],
    label="Recall",
    marker="o",
    color=color_recall,
)
ax1.plot(
    baseline_data["Shots"],
    baseline_data["F1"],
    label="F1 Score",
    marker="o",
    color=color_f1,
)
ax1.set_title("Performance of Baseline LLM")
ax1.set_xlabel("Number of Shots")
ax1.set_ylabel("Metric Value")
ax1.legend()
ax1.grid(True)
plt.show()

# plotting for LLM with emotion
fig2, ax2 = plt.subplots(figsize=(7, 5))
ax2.plot(
    emotion_data["Shots"],
    emotion_data["Precision"],
    label="Precision",
    marker="o",
    color=color_precision,
)
ax2.plot(
    emotion_data["Shots"],
    emotion_data["Recall"],
    label="Recall",
    marker="o",
    color=color_recall,
)
ax2.plot(
    emotion_data["Shots"],
    emotion_data["F1"],
    label="F1 Score",
    marker="o",
    color=color_f1,
)

ax2.set_title("Performance of LLM with Emotion")
ax2.set_xlabel("Number of Shots")
ax2.set_ylabel("Metric Value")
ax2.legend()
ax2.grid(True)
plt.show()

# %%
data = pd.read_csv('all_data.csv')
data['F1'] = pd.to_numeric(data['F1'], errors='coerce')

# making a list of colors for consistent colors across all graphs
colors = ['b', 'g', 'r', 'c', 'm']  # blue, green, red, cyan, magenta
color_dict = dict(zip(data['Approach'].unique(), colors))

plt.figure(figsize=(10, 6))
max_f1 = data['F1'].max()

# plots the lines
for approach in data['Approach'].unique():
    subset = data[data['Approach'] == approach]
    color = color_dict[approach]
    plt.plot(subset['Shots'], subset['F1'], label=approach, marker='o',
color=color)

#makes the graph/plot
plt.title('F1 Scores by LLM Type and Number of Shots')
plt.xlabel('Number of Shots')
plt.ylabel('F1 Score')
plt.legend()
plt.grid(True)
plt.ylim(0, max_f1 + 0.05)
plt.show()


# %%
data = pd.read_csv('all_data.csv')
data['F1'] = pd.to_numeric(data['F1'], errors='coerce')


plt.figure(figsize=(10, 6))

# making a list of colors for consistent colors across all graphs
colors = ['b', 'g', 'r', 'c', 'm']  # blue, green, red, cyan, magenta
color_dict = dict(zip(data['Approach'].unique(), colors))

# setting the max y-axis val
max_f1 = data['F1'].max()

# plot the lines
for approach in data['Approach'].unique():
    subset = data[data['Approach'] == approach].dropna()  # dropping NaN values
in the subset (this is for the 40-shot test that I don't have vals for due to
OpenAI API issue)
    color = color_dict[approach]
    plt.plot(subset['Shots'], subset['F1'], label=approach, marker='o',
color=color)

    # making the trend lines
    if len(subset['Shots']) >= 2:
        z = np.polyfit(subset['Shots'], subset['F1'], 1)
        p = np.poly1d(z)
        x_trend = np.linspace(min(subset['Shots']), max(subset['Shots']), 100)
        plt.plot(x_trend, p(x_trend), "--", color=color)

plt.title('F1 Scores by LLM Type and Number of Shots with Trend Lines')
plt.xlabel('Number of Shots')
plt.ylabel('F1 Score')
plt.legend()
plt.grid(True)
plt.ylim(0, max_f1 + 0.05)
plt.show()

# %%
```

```python
#this plots the f1 score of the LLM with Emotion based on the Emotion Level
(see report for details)

data = pd.read_csv('emotion.csv')
plt.figure(figsize=(8, 5))
plt.plot(data['Emotional Level'], data['F1 score'], marker='o', color='m')

# trend line code
z = np.polyfit(data['Emotional Level'], data['F1 score'], 1)
p = np.poly1d(z)
plt.plot(data['Emotional Level'], p(data['Emotional Level']), "m--")


plt.title('F1 Score by Emotional Level with Trend Line')
plt.xlabel('Emotional Level')
plt.ylabel('F1 Score')
plt.grid(True)
plt.show()
```

Bibliography

[1]Best practices for prompt engineering with the openai API. Accessed April 29, 2024. https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-the-openai-api.

[2]Zhang, Xiaodan, Nabasmita Talukdar, Sandeep Vemulapalli, Sumyeong Ahn, Jiankun Wang, Han Meng, Sardar Mehtab Bin Murtaza, et al. “Comparison of Prompt Engineering and Fine-Tuning Strategies in Large Language Models in the Classification of Clinical Notes.” medRxiv, January 1, 2024. https://www.medrxiv.org/content/10.1101/2024.02.07.24302444v1.full.

[3]Yin, Ziqi. Should we respect llms? A cross-lingual study on the influence of prompt politeness on LLM Performance. Accessed April 29, 2024. https://arxiv.org/html/2402.14531v1.

[4]Li, Cheng. ArXiv:2307.11760v7 [cs.CL] 12 Nov 2023. Accessed April 29, 2024. https://arxiv.org/pdf/2307.11760.pdf.