11-411/11-611 Homework Assignment 4: Entity Recognition

Lara Marinov - Spring 2024

Abstract

In this report, I experiment with multiple different methods of constructing demonstrations and prompts for the OpenAI large language model (LLM) to try to improve its performance on token classification. I first experimented with changing the number of shots on a baseline model, trying to discover which value of shots performed best. I then experimented with using three LLMs - Perplexity AI, ChatGPT 3.5, and Gemini - to create modified prompts for the OpenAI LLM. I conclude that across both baseline approaches and approaches with modified prompts, using 40 shots and the baseline prompt worked best with an F1 score of 0.361. Of the three modified prompt approaches, ChatGPT's "fill-in-the-blank" prompt style worked best with an F1 score of 0.302.

Part 1: Methodology

Background and Justification

Prompt engineering has been an active area of NLP research, focusing on techniques to improve the performance of large language models (LLMs) on specific tasks by carefully crafting prompts. Many formatting and high-level techniques for prompt engineering have been previously explored like few-shot prompting, where the prompt includes examples of the desired task to guide the LLM generation The New Stack (2023), and chain-of-thought prompting, where the LLM is instructed to break down its reasoning process into multiple steps Chen et al. (2023).

Along with high level formatting techniques, other research has been done on smaller scale changes which also improve LLM performance such as using affirmative directives, incorporating examples, assigning roles to LLMs, and using output primers Bsharat et al. (2024).

The techniques mentioned above have shown promise in improving LLM performance, but they primarily focus on generating highly likely or factual outputs. There has also been research into generating relevant, but less probable prompts, which can be valuable in decision-making scenarios where considering alternative possibilities is important Tang et al. (2023).

While significant research has explored multiple prompt engineering techniques, little research exists which has explored utilizing LLMs themselves as auxiliary tools within the overall prompt engineering process. Traditionally, research has focused on improving and refining human-crafted prompts for direct LLM interaction. This short report aims to investigate the potential of LLMs to contribute to the development of effective prompts for other LLMs.

Section 1.1: Baseline Experimentation

To start, I prompted the OpenAI LLM with the baseline format given to us in the notebook which can be seen in **Figure 1**. By varying only the number of shots and keeping the same prompt structure, I could understand the effect of training data size on the model's performance in an isolated environment. This allowed me to assess how OpenAI LLM's ability to identify and classify entities changed as the amount of labeled examples provided increased. My subsequent runs then introduced variations to the prompt structure itself, building upon the basic understanding of the model's behavior using the baseline format. The initial evaluation served as a sort of control experiment, establishing a benchmark for how the OpenAI LLM performed with a fixed prompt format before introducing additional complexities.

The LLM performed reasonably well, with the F1 score increasing as the number of shots increased. The highest F1 score was 0.361, which was achieved using a 40-shot approach. More results can be seen in **Figure 4** in Section 2. I wanted to run additional experiments, increasing the number of shots to 100, but I also recognized that it was important to experiment with different prompt structures before I ran out of credits. Given the resource limitation and lack of information regarding how many credits I had left, I decided to switch focus to changing the prompt structure and revisit the impact of higher shot sizes if I still had credits. I unfortunately ran out of credits before I could return to run with increased shot values. **Section 3.4** discusses the limitations encountered and how they affected the rest of my report in more detail. With more credits, I would have increased the shot size to at least 100, and if performance continued to improve, I may have decided to further increase the shot size to see how performance would change.

```
messages = [
    {'role': SYSTEM_STR, MSG_STR:
     """You will be given input text containing different types of entities that you will label.
     This is the list of entity types to label: Deity, Mythological_king, Cretaceous_dinosaur, Aquatic_mammal, Aquatic_animal, Goddess.
     Label the enities by surrounding them with tags like '<Cretaceous_dinosaur> Beipiaognathus </Cretaceous_dinosaur>'."""
     },
    {'role': USER_STR, MSG_STR: """Text: Once paired in later myths with her Titan brother Hyperion as her husband, mild-eyed Euryphaessa, the far-shining one of the Homeric Hymn to Helios, was said to be the mother of Helios (the Sun), Selene (the Moon), and Eos (the Dawn)."""},
    {'role': SYSTEM_STR, MSG_STR: """Labels: Once paired in later myths with her Titan brother <Deity> Hyperion </Deity> as her husband, mild-eyed Euryphaessa, the far-shining one of the Homeric Hymn to Helios, was said to be the mother of Helios (the Sun), <Goddess> Selene </Goddess> (the Moon), and <Goddess> Eos </Goddess> (the Dawn)."""},
    {'role': USER_STR, MSG_STR: """Text: From her ideological conception, Taweret was closely grouped with (and is often indistinguishable from) several other protective hippopotamus goddesses: Ipet, Reret, and Hedjet.\nLabels: """}
]
```

Figure 1: The baseline prompt format provided in the notebook

Section 1.2: Modified Prompts Experimentation

In an effort to explore a unique prompt engineering technique, I investigated the feasibility of utilizing an LLM to generate prompt structures for the OpenAI LLM backend. This approach involved getting prompt structures from three well-known conversational LLMs: Perplexity AI, ChatGPT, and Gemini. All three models received the exact same sequence of two prompts. The first prompt, seen in **Figure 2**, outlined the task and the desired entity tags for classification. I included the baseline prompt into the message so that the models could understand the context of the current prompts.

```
I am working on prompt engineering for an LLM which is doing token classification. This is the current structure of the prompt:

chat_history =[{'role': SYSTEM_STR, MSG_STR: '''You will be given input text containing different types of entities that you will label. This is the list of entity types to label: Deity, Mythological_king, Cretaceous_dinosaur, Aquatic_mammal, Aquatic_animal, Goddess. Label the enities by surrounding them with tags like '<Cretaceous_dinosaur> Beipiaognathus </Cretaceous_dinosaur>'.'''},
{'role': USER_STR, MSG_STR: '''Text: Once paired in later myths with her Titan brother Hyperion as her husband, mild-eyed Euryphaessa, the far-shining one of the Homeric Hymn to Helios, was said to be the mother of Helios (the Sun), Selene (the Moon), and Eos (the Dawn).'''},
{'role': SYSTEM_STR, MSG_STR: '''Labels: Once paired in later myths with her Titan brother <Deity> Hyperion </Deity> as her husband, mild-eyed Euryphaessa, the far-shining one of the Homeric Hymn to Helios, was said to be the mother of Helios (the Sun), <Goddess> Selene </Goddess> (the Moon), and <Goddess> Eos </Goddess> (the Dawn).'''}]

Write a few options for different prompts I could give to the LLM
```

Figure 2: The prompt given to the three LLMs

Each LLM gave 3-4 suggestions for approaches with example messages. All three responses shared some similarities in the approaches they suggested, but there were also considerable differences. Each LLM suggested basic methods such as zero shot and few shot prompting, which I already covered in the experiments with the baseline prompts. To decide which prompt structure suggestion I would use from each model, I chose the suggestion most dissimilar to the responses from the other LLMs. I then prompted each LLM once more with the message in **Figure 3**, only switching out the text with {"examples with explanations", "fill in the blank", 'error-augmented prompts'} for the respective LLM.

Section 1.2.1: Perplexity AI – Examples with explanations

The "examples with explanations" prompt engineering technique is a powerful, intuitive approach where examples are provided along with correct answers and explanations. The core idea is to influence the LLM by showing it examples of the desired task along with the correct outputs and, crucially, explanations for why those outputs are correct Weng (2023). This technique helps the LLM better understand the task and the reasoning required to generate the desired classification. By seeing the explanations of the correct answers, the model can learn the patterns, logic, and context behind the inputs and can apply that knowledge to future inputs.

For the specific entity recognition task of this assignment, the input text and labels would be the same as in the baseline approach. The explanation added to each pair of input text and labels would include a description of why the specific special tags were chosen for each token. For example, a possible explanation could describe that "Athena" is a Greek goddess which is why she is tagged with the "Goddess" tag. **Appendix Section A1** shows the complete output from Perplexity AI which was used as the precedent to create the rest of the message triples sent to the OpenAI backend.

Section 1.2.2: ChatGPT – Fill in the blank

The "fill-in-the-blank" prompt engineering technique for token classification is a way to structure input prompts so that the LLM is asked to identify and label specific entities within a given text by replacing blank spaces with the appropriate tags Liu et al. (2021).

Instead of providing the labeled text associated with each input directly, the prompt presents the text with blank spaces where the HTML tags should be inserted: < __ > Hyperion < __ >.

By presenting the task in this "fill-in-the-blank" format, the LLM is prompted to identify the relevant entities within the text and replace the blank spaces with the appropriate tags. One significant potential advantage of this structure is that the LLM, instead of passively receiving the labeled text, is actively engaged in the task of identifying and labeling the tokens. The hope is that this active involvement leads the LLM to better understand and remember the context and goal of the task.

There is also an extension of the "fill-in-the-blank" format where, in addition to examples with blank tags, a few examples of correctly labeled texts are also included. These correct examples could possibly improve the LLM's performance by helping it learn the desired output format of the entity labeling.

Section 1.2.3: Gemini – Error-augmented prompts

The "error-augmented prompts" prompt engineering technique for token classification involves intentionally introducing errors, such as filler words, spelling mistakes, or inconsistent capitalization, into the prompts provided to the LLM.

By including different types of errors in the input messages, the model is forced to learn to identify and extract the relevant entities and their tags despite distractions and irregularities being present in the input. This can be particularly useful for real-world applications where the input text may not always be perfectly formatted or error-free Eliot (2023).

By training the model on prompts with various types of errors, it can learn to extract relevant information better and it can handle a wider range of inputs. The hope is that because the model becomes less sensitive to minor formatting issues, typos, or inconsistencies in the input text, it is able to focus its attention on the accuracy of the entity tags. The model may develop a better understanding of the context, which would further help it differentiate relevant entities.

```
Expand more on the <see above> approach and give an example prompt in code
using a structure like the one I provided in my previous message.
```

Figure 3: The follow-up prompt given to each LLM to receive uniform prompts to feed to the OpenAI backend

Part 2: Experimental Results

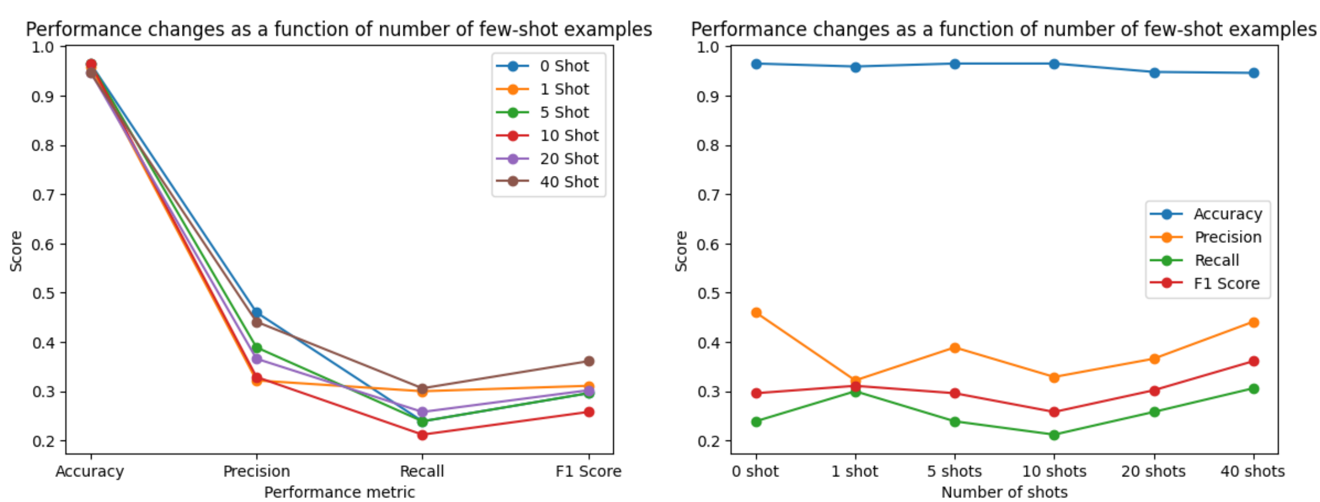
Approach	Shots	Accuracy	Precision	Recall	F1
Finetune BERT	-	0.948	0.410	0.509	0.454
Zero-shot LLM Baseline	0	0.965	**0.460**	0.239	0.296
One-shot LLM Baseline	1	0.959	0.322	0.300	0.311
Few-shot LLM Baseline	5	0.965	0.389	0.239	0.296
Few-shot LLM Baseline	10	**0.965**	0.329	0.212	0.258
Few-shot LLM Baseline	20	0.948	0.366	0.258	0.302
Few-shot LLM Baseline	40	0.946	0.441	**0.306**	**0.361**
Perplexity AI suggestion	5	0.964	0.363	0.242	0.298
ChatGPT suggestion	5	0.956	0.370	0.256	0.302
Gemini suggestion	5	0.958	0.347	0.263	0.290

Figure 4: Experimental results table, best result (excluding BERT) in each column is shown in **bold**

As discussed in **Section 1**, I chose to experiment with the feasibility of utilizing an LLM to generate prompt structures for the OpenAI LLM for the token classification task. My experimental results table is displayed above. The first observation to note is that the approach with the highest F1 score was using the LLM baseline with 40 shots. This result is not surprising considering the nature of the specific labeling we are asking the model to perform. Although, some discussion suggests that larger models do not need as many shots because they have seen more data in training and can thus generalize better to a wide variety of tasks (Reynolds & McDonell (2021); Weird_Foundation5044 (2023)).

The second interesting result to note is that the accuracy across all runs is far higher than the score on any other metric, regardless of approach. This result is discussed in depth in **Section 3** of this report. All runs with modified prompt structures were done using 5 shots. More runs were planned with increased shots, but the restricted number of API credits limited the number of runs I was able to accomplish. These limitations are discussed further in **Section 3.4**.

Performance changes



(a) Performance changes plotted by metric

(b) Performances changes plotted by shots

Figure 5: Performances changes with baseline prompt structure and changes to number of shots

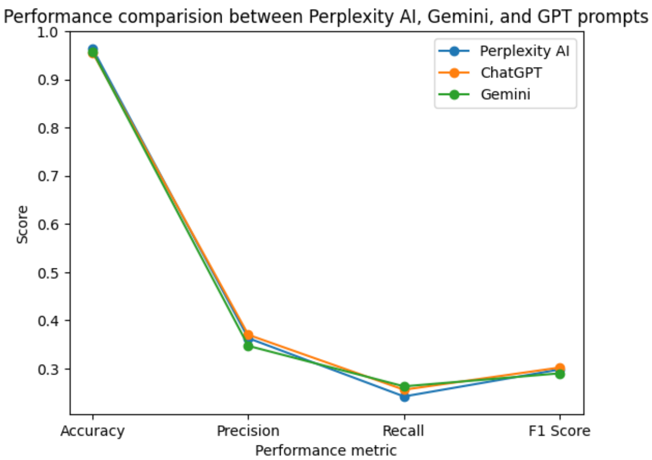


Figure 6: Performances changes plotted by metric

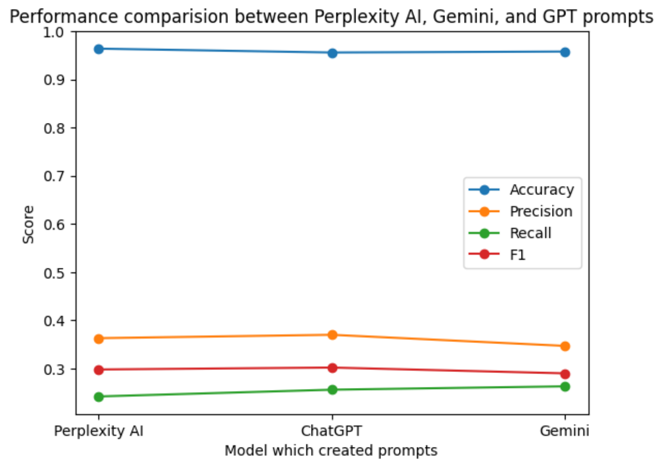


Figure 7: Performances changes plotted by run type

Figure 8: Performances changes with prompt structure modification

Figure 5 shows how the performance changes as a function of number of few-shot examples. Once again, accuracy is significantly higher across all of the shots, as can be seen in **Figure 5b**. The run with 40 shots performed better across all four metrics, although the zero-shot run also had very similar precision to the 40-shot run. **Figures 6 - 8** show the same performance changes, but on the runs with modified prompt structures.

The performances across the different prompt structures in comparison to each other are more similar than the performances with different shots in comparison to each other. The "fill in the blank" prompt style from ChatGPT marginally outperformed the two styles from Perplexity AI and Gemini. These results are discussed in more depth in **Section 3**.

Part 3: Analysis and Discussion

		'O'	'Aquatic_animal'	'Aquatic_mammal'	'Cretaceous_dinosaur'	'Deity'	'Goddess'	'Mythological_king'
Train set	Raw Number	161563	1304	714	450	1092	997	313
	Raw Number (B | I)	–	739 | 565	451 | 263	438 | 12	969 | 123	852 | 145	247 | 66
	Fraction of total	0.9699	0.0081	0.0044	0.0028	0.0068	0.0062	0.0019
	Fraction of category (B | I)	–	0.5667 | 0.4333	0.6317 | 0.3683	0.9733 | 0.0267	0.8874 | 0.1126	0.8546 | 0.1454	0.7891 | 0.2109
Dev set	Raw Number	13311	101	49	39	128	78	16
	Raw Number (B | I)	–	60 | 41	34 | 15	36 | 3	114 | 14	68 | 10	14 | 2
	Fraction of total	0.9700	0.0074	0.0036	0.0028	0.0093	0.0057	0.0012
	Fraction of category (B | I)	–	0.5941 | 0.4059	0.6939 | 0.3061	0.9231 | 0.0833	0.8906 | 0.1094	0.8718 | 0.1282	0.875 | 0.125
Both sets	Raw Number	170004	1405	763	489	1220	1075	329
	Raw Number (B | I)	–	799 | 606	485 | 278	474 | 15	1083 | 137	920 | 155	261 | 68
	Fraction of total	0.9699	0.0080	0.0044	0.0028	0.0069	0.0061	0.0019
	Fraction of category (B | I)	–	0.5687 | 0.4313	0.6356 | 0.3644	0.9693 | 0.0307	0.8877 | 0.1123	0.8558 | 0.1442	0.7933 | 0.2067

Figure 9: Breakdown of entity type tag distribution across dev and training datasets

Section 3.1 Analysis The analysis of the experimental results revealed that while the performance improvement for modifications to the prompt structure were relatively minor, increasing the number of shots consistently improved model performance. This suggests that the model benefits from a larger dataset, even with marginally different prompts.

However, analyzing the breakdown of tags in both datasets in **Figure 9** and **Figures 10 - 12** revealed that the datasets are overwhelmingly skewed towards the 'O' tag. As observed in **Section 2**, the accuracy across all runs – both more shots and modified prompts – was significantly higher than the rest of the metrics, which is likely a direct cause of this imbalance; even the most simplistic model that predicts only 'O' for every token could achieve a deceptively high accuracy score.

This observation also raised the question: how do the modified prompt structures possibly reduce this tag imbalance problem? One plausible explanation is that the different prompt structures can help the model learn that the task is not just about identifying each token as the 'O' tag but also recognizing and labeling other entity types based on contextual information. By exposing the model to various prompt structures with examples and explanations, it may capture the nuances and patterns required for accurate tag recognition better, rather than defaulting to a naive 'O' prediction strategy.

To further investigate the effectiveness of the modified prompts, I thought it would be insightful to analyze the performance metrics across the different prompt formats, broken down by entity tag type. I thought that this granular analysis could reveal valuable insights into which prompting techniques are more adept at identifying specific entity types, rather than simply relying on the majority 'O' class predictions consistently.

Figures 10 - 12 shows the breakdown of the tags in the dev and training sets, excluding 'O'. Clearly, some labels are more common than others: the 'Deity' and 'Goddess' tags appear more frequently than tags like 'Mythological_king'. So, if a model is good at predicting the 'Deity' and 'Goddess' tags, we would expect it to perform better overall, even if it's worse at predicting many of the other tags. This suggestion is supported by the performance of the 40-shot run, which had the highest F1 score on the 'Deity' and 'Goddess' tags (as seen in **Figures 16 and 17**) and the highest overall F1 score, even though it was not consistently the best at predicting every tag. These results could suggest that to enhance overall model performance in the future, prompts could strategically emphasize the less frequently encountered tags. This approach would aim to address the potential bias towards frequent tags and it would encourage the model to focus on tags where its learning can be further optimized.

Figures 13 - 18 also reveal that there was not one run type which had better performance across all tags. Indeed, quite the opposite is true as each of the four run types: 40-shot, Perplexity AI, ChatGPT, and Gemini had at least one tag for which they performed the best (as measured by the F1 score). This suggests that combining the prompting style used in the Perplexity AI, ChatGPT, and Gemini runs and increasing the numbers of shots (possibly to even more than 40) could further improve the model's performance.

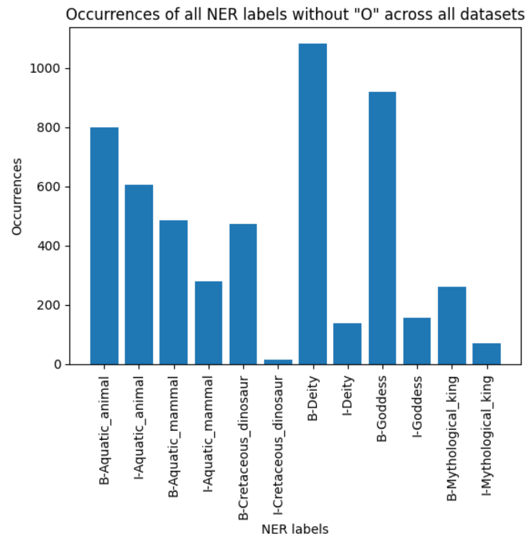


Figure 10: Distribution of tags across both the dev and training sets; generation code in Appendix Section A5

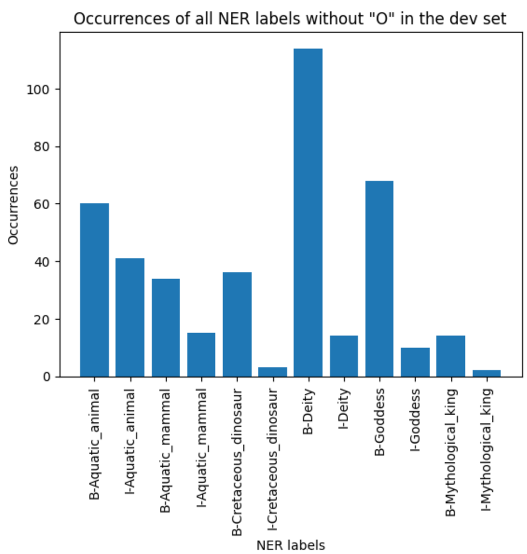


Figure 11: Distribution across the dev set; generation code in Appendix Section A5

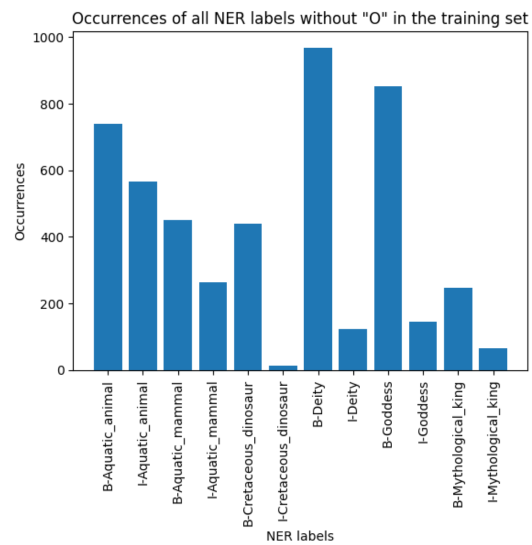


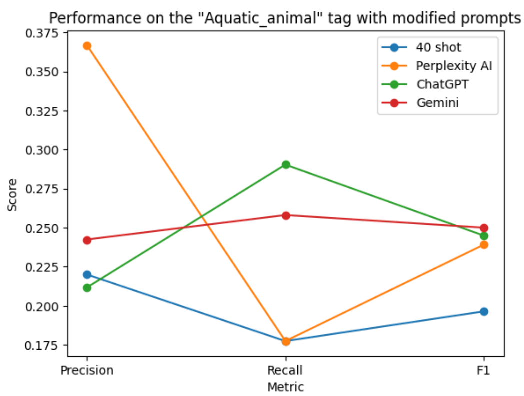
Figure 12: Distribution across the training set; generation code in Appendix Section A5

Further, if a particular prompt structure consistently outperforms others in recognizing and labeling rare entity types, such as "Mythological_king" or "Cretaceous_dinosaur," it could indicate that the prompt structure is better at capturing the contextual cues and patterns associated with those entities. Identifying the prompt structures which result in better performance on rare tags is crucial. These prompt structures could be interwoven with ones that perform better on common tags, possibly resulting in models that predict both common and rare tags more accurately.

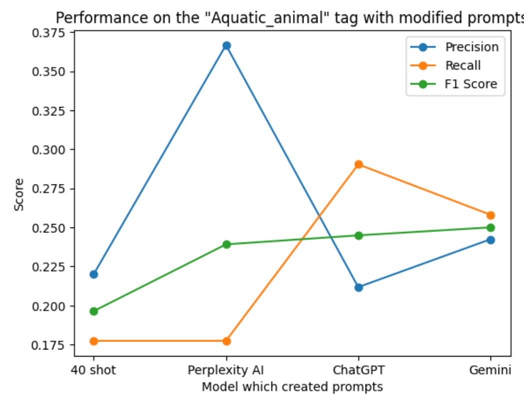
Section 3.2 Future plans It would be interesting to further investigate the potential of leveraging LLMs as auxiliary prompt engineering tools. The research could explore the efficacy of LLMs in generating effective prompt structures along with compiling a comparative analysis of the strengths and weaknesses of these LLM-generated prompts.

Section 3.3 Conclusion In this short report, I explored different methods of constructing demonstrations and prompts for the OpenAI LLM to try to improve its performance on token classification. I experimented both with a baseline prompt structure, varying only the number of shots, and with three different modified prompt structures: (1) examples with explanations, (2) fill in the blank, and (3) error-augmented prompts. These three approaches were generated using Perplexity AI, ChatGPT, and Gemini respectively. I concluded that the 40-shot approach worked the best because it had the highest F1 score, but also aknowledged that each run type performed better than the others at identifying at least one token type. With further experimentation, I would have liked to see how a combination of prompt styles changed the performance of the LLM.

Section 3.4 Limitations As previously noted, resource limitations in the form of available credits significantly impacted the scope of the experiments. My original plan included the following runs: {0, 1, 5, 10, 20, 40, 100, 200} shots with original prompt structure, {0, 1, 5, 10, 20, 40, 100, 200} shots with a {fill in the blank, examples with explanations, error-augmented prompts} prompt structure, and an undetermined number of runs combining different numbers of shots and multiple prompt techniques. Unfortunately, due to credit constraints, executing the full range of experiments was not feasible. I used the OpenAI backend on each of the runs I executed. Using the Cohere platform was a potential way to expand expanding the run list, but my concerns regarding potential bias introduced by utilizing different backends led me to decide that it was best to maintain consistency for the sake of result purity. Additionally, I could have paid for my own API key, but without knowing how much cost that could incur, I decided against this approach.

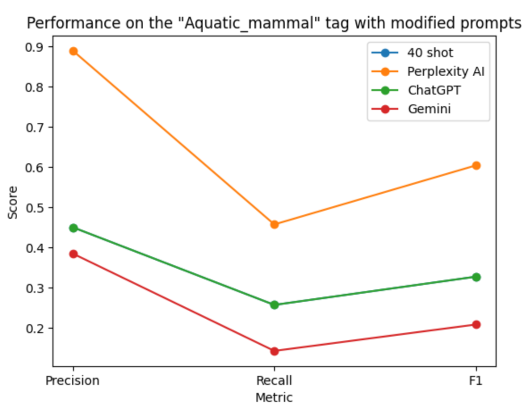


(a) Performance changes plotted by metric

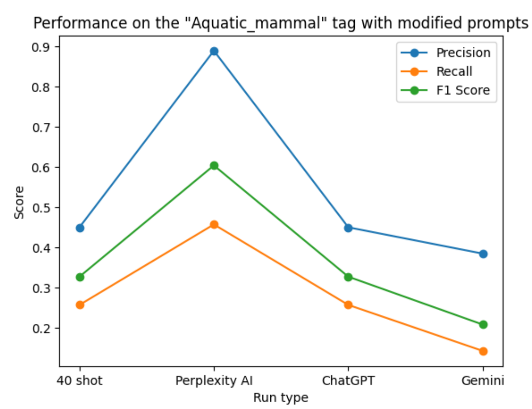


(b) Performance changes plotted by structure

Figure 13: Performance changes on the “Aquatic_animal” tag; generation code in Appendix Section A5

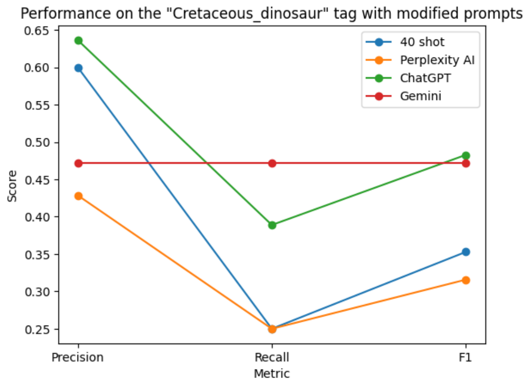


(a) Performance changes plotted by metric; NOTE: the 40 shot run and the ChatGPT run had the exact same metrics, which is why there are only 3 visible lines

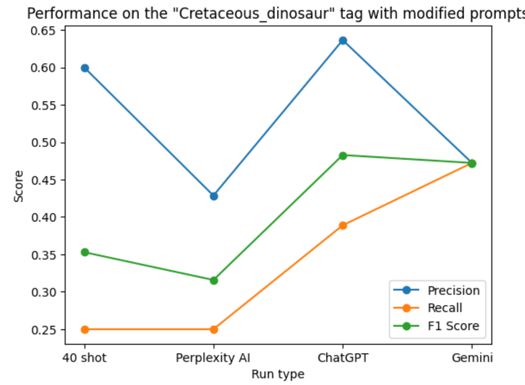


(b) Performance changes plotted by structure

Figure 14: Performance changes on the “Aquatic_mammal” tag; generation code in Appendix Section A5

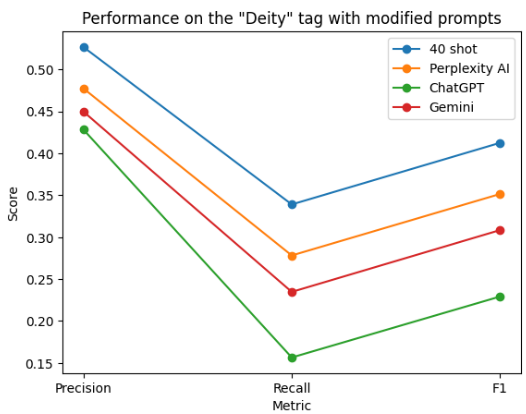


(a) Performance changes plotted by metric

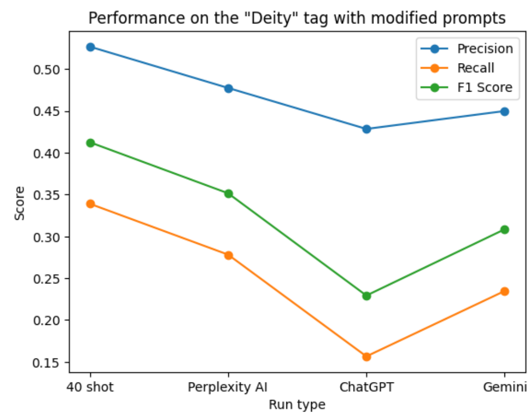


(b) Performance changes plotted by structure

Figure 15: Performance changes on the "Cretaceous_dinosaur" tag; generation code in Appendix Section A5

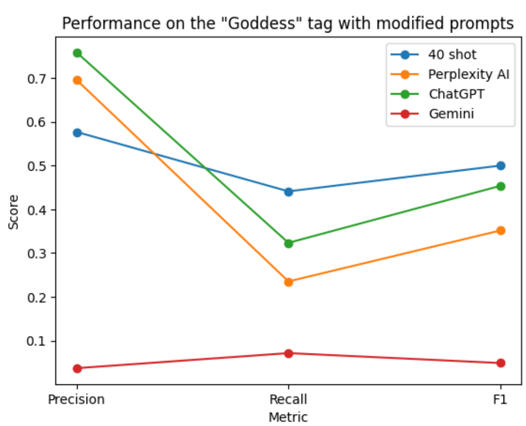


(a) Performance changes plotted by metric

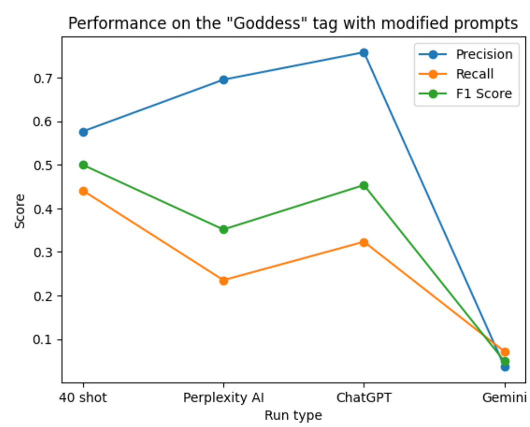


(b) Performance changes plotted by structure

Figure 16: Performance changes on the "Deity" tag; generation code in Appendix Section A5

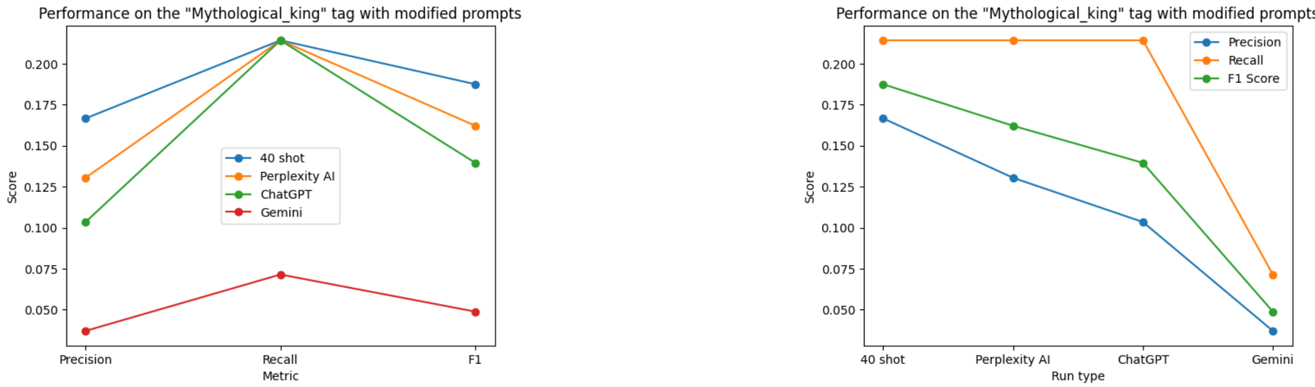


(a) Performance changes plotted by metric



(b) Performance changes plotted by structure

Figure 17: Performance changes on the "Goddess" tag; generation code in Appendix Section A5



(a) Performance changes plotted by metric

(b) Performance changes plotted by structure

Figure 18: Performance changes on the "Mythological_king" tag; generation code in Appendix Section A5

Appendix: Code

Section A0: Original prompt structure code

```python
# The baseline prompt structure format
messages = [
    {'role': SYSTEM_STR, MSG_STR:
     """You will be given input text containing different types of entities that you will label.
     This is the list of entity types to label: Deity, Mythological_king, Cretaceous_dinosaur, Aquatic_mammal, Aquatic_animal, Goddess.
     Label the enities by surrounding them with tags like '<Cretaceous_dinosaur> Beipiaognathus </Cretaceous_dinosaur>'."""
     },
     {'role': USER_STR, MSG_STR: """Text: Once paired in later myths with her Titan brother Hyperion as her husband, mild-eyed Euryphaessa, the far-shining one of the Homeric Hymn to Helios, was said to be the mother of Helios (the Sun), Selene (the Moon), and Eos (the Dawn)."""},
     {'role': SYSTEM_STR, MSG_STR: """Labels: Once paired in later myths with her Titan brother <Deity> Hyperion </Deity> as her husband, mild-eyed Euryphaessa, the far-shining one of the Homeric Hymn to Helios, was said to be the mother of Helios (the Sun), <Goddess> Selene </Goddess> (the Moon), and <Goddess> Eos </Goddess> (the Dawn)."""},
     {'role': USER_STR, MSG_STR: """Text: From her ideological conception, Taweret was closely grouped with (and is often indistinguishable from) several other protective hippopotamus goddesses: Ipet, Reret, and Hedjet.\nLabels: """}
]

# Returns the content of the example
def get_message(example):
    return """{}""".format(example['content'])

# A function which takes an example from the dataset as input, and returns a string that has tagged the text with labels in the given HTML-style format.
def convert_bio_to_prompt(example):
    message = ""
    current_label = None
    # All we need is the token and its corresponding ner_string, we can ignore the rest of the information stored for each example
    for token, ner_string in zip(example['tokens'], example['ner_strings']):
        if ner_string != "O":
            # Extract the label from the token
            label = ner_string.split('-')[-1]
            if label != current_label:
                if current_label:
                    # Finishing the tagged entity, so add `/` to indicate the closing tag
                    message += "</{}> ".format(current_label)
                # Otherwise, we are at the beginning of the tagged entity
                message += "<{}> ".format(label)
                current_label = label
            message += token.split('-')[0] + " "
        # If the current token is not a named entity but there was just one
        elif current_label:
            message += "</{}> ".format(current_label)
            current_label = None
        else:
            message += token.split('-')[0] + " "
    # If the example ends with a named entity
    if current_label:
        message += "</{}>".format(current_label)
    return message


```

```python
# 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.
def get_chat_history(shots, dataset, entity_types_list, convert_bio_to_prompt_fn):
    chat_history = []

    # System message
    system_message = {
        'role': SYSTEM_STR, MSG_STR: "You will be given input text containing different types of entities that you will label. This is the list of entity types to label: {}. Label the entities by surrounding them with tags like '<{}> Entity </{}>'.".format(
            ", ".join(entity_types_list), list(entity_types_list)[0], list(entity_types_list)[0])
    }
    chat_history.append(system_message)

    for i in range(shots):
        example = dataset[i]
        # Get the text from each sample in the dataset
        user_message = {
            'role': USER_STR, MSG_STR: """Text: {}""".format(get_message(example))
        }
        chat_history.append(user_message)

        # Add the labels to the sample
        system_message = {
            'role': SYSTEM_STR, MSG_STR: """Labels: {}""".format(convert_bio_to_prompt_fn(example))
        }
        chat_history.append(system_message)

    return chat_history

# Function to handle punctuation which is not correctly attached to the closest token to the left
def move_punctuation_left(text):
    # Define regular expression pattern to match punctuation followed by spaces
    pattern = re.compile(r'\s*([.,:;!?])\s*')
    matches = re.finditer(pattern, text)
    modified_text = ''
    # Position of last match
    last_match_end = 0
    for match in matches:
        punctuation = match.group(1)
        # Get the start and end index of the match
        start, end = match.span()
        # Move the punctuation mark to touch the closest word on the left
        modified_text += text[last_match_end:start].rstrip() + punctuation
        # Update the position of the last match
        last_match_end = end
    # Append the remaining text after the last match
    modified_text += text[last_match_end:]
    return modified_text

def convert_response_to_bio(response):
    labels = []
    # bool: are we inside of a tag, bool: are we at the first position within the tag, bool: was the previous position in the tag, str: the name of the tag)
    (inside, first, prev, tag) = (False, False, False, None)
    for word in response.split():
        # Ignore the starting labels
```

```python
        if "Labels:" in word:
            continue
        # At the end of a tag
        if "</" in word:
            rest = re.sub(r'<.+?>', '', word)
            tag = (word.split('</'))[1].split('>')[0]
            if (word[-1] != '>' and any(char in string.punctuation for char in word.split('>')) and word.split('<')[0] == ''):
                # Reset everything because we are now outside of the tag
                (inside, first, prev, tag) = (False, False, False, None)
                continue;
            # Inside of the tag
            elif (rest != '' and prev):
                labels.append("I-" + tag)
            # At the beginning
            elif (rest != '' and not prev):
                labels.append("B-" + tag)
            (inside, first, prev, tag) = (False, False, False, None)
            continue
        # At the beginning of a tag
        elif "<" in word:
            (inside, first, prev, tag) = (True, True, False, (word.split('<'))[1].split('>')[0])
            rest = re.sub(r'<.+?>', '', word)
            # This tag has multiple tokens that it is applied to
            if (rest != ''):
                labels.append("B-" + tag)
                first = False
                prev = True
            continue
        if inside:
            if first:
                labels.append("B-" + tag)
                first = False
                prev = True
            else:
                labels.append("I-" + tag)
        else:
            labels.append("O")

    # Ensure that lonely punctuation marks are handled properly
    tokens_pattern = re.compile(r'<[^>]+>')
    no_tags = re.sub(tokens_pattern, '', response)
    no_punct_spaces = re.sub(r'\s([?.!"](?:\s|$))', r'\1', no_tags)
    no_punct_spaces2 = move_punctuation_left(no_tags)

    tokens = re.sub(r'([{}])'.format(r'\s*.,:;!?\s*'),r'\1 ', no_punct_spaces2).split()
    if "Label" in tokens[0]:
        tokens = tokens[1:]

    return labels, tokens
```

Section A1: Perplexity AI "examples with explanations" prompt structure

The full output created by Perplexity AI as an example input for the "examples with explanations" prompt structure. See **Figure 2** and **Figure 3** in **Section 1** for the prompts given to ChatGPT to create this prompt structure.

```python
messages = [
{'role': SYSTEM_STR, MSG_STR:
"""You will be given input text containing different types of entities that you will label. This is the list of entity types to label: Deity, Mythological\_king, Cretaceous\_dinosaur, Aquatic\_mammal, Aquatic\_animal, Goddess. Label the entities by surrounding
```

```
    them with tags like '<Cretaceous\_dinosaur> Beipiaognathus </Cretaceous\_dinosaur>'.
    Here are some examples with explanations:"""
    },
    {'role': USER_STR, MSG_STR:
    """Example 1:
    Input text: 'Poseidon, the god of the sea, ruled over the oceans and all creatures that
    dwelled within them, including dolphins and whales."}
    {'role': SYSTEM_STR, MSG_STR:
    Labels: "<Deity> Poseidon </Deity>, the god of the sea, ruled over the oceans and all
    creatures that dwelled within them, including <Aquatic\_mammal> dolphins </Aquatic\
    _mammal> and <Aquatic\_mammal> whales </Aquatic\_mammal>.'
    Explanation: Poseidon is a deity, dolphins and whales are aquatic mammals."""
    },
    {'role': USER_STR, MSG_STR:
    """Example 2:
    Input text: 'The infraclassis Carinacea includes most living species of regular sea
    urchin, and fossil forms going back as far as the Triassic.'}
    {'role': SYSTEM_STR, MSG_STR:
    Labels: 'The infraclassis Carinacea includes most living species of regular <Aquatic\
    _animal> sea urchin </Aquatic\_animal>, and fossil forms going back as far as the
    Triassic.'
    Explanation: A sea urchin is an aquatic animal."""
    }
    {'role': USER_STR, MSG_STR:
    Now, try labeling the following input text:
    Input text: "From her ideological conception, Taweret was closely grouped with (and is
    often indistinguishable from) several other protective hippopotamus goddesses: Ipet,
    Reret, and Hedjet."}

```

The change from the baseline code to generate this structure of prompt is in `get_chat_history`. This version extracts an explanation from the labeling and appends it to the system message.

```
def add_explanation():
    # Use regular expression to find content within <tag> tags
    match = re.search(r'<(.*?)>(.*?)<\/\1>', text)
    if match:
        # Extract tag name and content within the tags
        tag_name = match.group(1)
        content = match.group(2)
        # Create the explanation
        return f"Explanation: A {content.lower()} is a {tag_name}."
    else:
        return "No explanation found."

def get_chat_history(shots, dataset, entity_types_list, convert_bio_to_prompt_fn):
    # System message
    system_message = {
        'role': SYSTEM_STR, MSG_STR: """You will be given input text containing different
    types of entities that you will label. This is the list of entity types to label: {}.
    Label the entities by surrounding them with tags like '<{}> Beipiaognathus </{}>'. Here
    are some examples with explanations:"""
    }.join(entity_types_list), list(entity_types_list)[0], list(entity_types_list)[0])
    chat_history.append(system_message)

    for i in range(shots):
        example = dataset[i]
        user_message = {
            'role': USER_STR, MSG_STR: """Text: {}""".format(get_message(example))
        }
        chat_history.append(user_message)
```

```python

        labeled = convert_bio_to_prompt_fn(example)
        system_message = {
            'role': SYSTEM_STR, MSG_STR: """Labels: {} {}""".format(labeled, add_explanation(labeled))
        }
        chat_history.append(system_message)

    return chat_history
```

Section A2: ChatGPT “fill in the blank” prompt structure

This is the example prompt generated by ChatGPT using the “fill in the blank” prompt structure. See **Figure 2** and **Figure 3** in **Section 1** for the prompts given to ChatGPT to create this prompt structure.

```python
messages = [
    {'role': SYSTEM_STR, MSG_STR:
     """Identify and label the entities in the following text. Replace the blank space with the appropriate tags. This is the list of entity types to label: Deity, Mythological_king, Cretaceous_dinosaur, Aquatic_mammal, Aquatic_animal, Goddess. Label the entities by replacing the blank space with tags like '<Deity> Hyperion </Deity>'."""
     },
     {'role': USER_STR, MSG_STR: """Text: Once paired in later myths with her Titan brother Hyperion as her husband, mild-eyed Euryphaessa, the far-shining one of the Homeric Hymn to Helios, was said to be the mother of Helios (the Sun), Selene (the Moon), and Eos (the Dawn)."""},
     {'role': SYSTEM_STR, MSG_STR: """Labels: Once paired in later myths with her Titan brother <__> Hyperion <__> as her husband, mild-eyed Euryphaessa, the far-shining one of the Homeric Hymn to Helios, was said to be the mother of Helios (the Sun), <__> Selene <__> (the Moon), and <__> Eos <__> (the Dawn)."""}
]

message = f"""Text: From her ideological conception, Taweret was closely grouped with (and is often indistinguishable from) several other protective hippopotamus goddesses: Ipet, Reret, and Hedjet.
Labels: """
```

The most important change to the baseline code to create the ChatGPT fill-in-the-blank style prompt is to the function `convert_bio_to_prompt`. Instead of adding the proper entity tags, we add “_” everywhere.

```python
# The modified convert_bio_to_prompt code to generate prompts of the same structure as the example created by ChatGPT
def convert_bio_to_prompt(example):
    message = ""
    # The labels will all be replaced with "_" to create the fill-in-the-blank structure
    current_label = "_"
    for token, ner_string in zip(example['tokens'], example['ner_strings']):
        if ner_string != "O":
            label = ner_string.split('-')[-1]  # Extracting the entity label from the token
            if label != current_label:
                if current_label:
                    message += "<{}> ".format(current_label)
                # Add "_" instead of the actual tag to create fill-in-the-blank prompt
                message += "<{}> ".format("_")
                current_label = "_"
            message += token.split('-')[0] + " "
        # If the current token is not a non-"O" tag but there was just one
        elif current_label:
            # Once again add "_" instead of actual tag to create fill-in-the-blank prompt
            message += "<{}> ".format("_")
            current_label = None
```

```python
        else:
            message += token.split('-')[0] + " "

    # Check if the ending is a non-"O" tag
    if current_label:
        message += "<{}>".format(current_label)
    return message
```

Section A3: Gemini "error-augmented prompts" prompt structure

This is the example prompt generated by Gemini using the 'error-augmented prompts" prompt structure. See **Figure 2** and **Figure 3** in **Section 1** for the prompts given to ChatGPT to create this prompt structure.

```python
    original_text = "Once paired in later myths with her Titan brother Hyperion as her husband, mild-eyed Euryphaessa, the far-shining one of the Homeric Hymn to Helios, was said to be the mother of Helios (the Sun), Selene (the Moon), and Eos (the Dawn)."
    original_label = "Once paired in later myths with her Titan brother <Deity> Hyperion </Deity> as her husband, mild-eyed Euryphaessa, the far-shining one of the Homeric Hymn to Helios, was said to be the mother of Helios (the Sun), <Goddess> Selene </Goddess> (the Moon), and <Goddess> Eos </Goddess> (the Dawn)."

    error_text_1 = original_text.replace("Hyperion", "Hipyrion").replace("Selene", "sELenE")
    err_label_1 = original_label.replace("Hyperion", "Hipyrion").replace("Selene", "sELenE")

    clean_prompt = {'role': USER_STR, MSG_STR: f"""Text: {original_text}"""}
    clean_response = {'role': SYSTEM_STR, MSG_STR: f"""Labels: {original_label}"""}

    error_prompt = {'role': USER_STR, MSG_STR: f"""Text: {error_text_1}"""}
    error_response = {'role': SYSTEM_STR, MSG_STR: f"""Labels: {err_label_1}"""}

    messages = [
        {'role': SYSTEM_STR, MSG_STR:
        """You will be given input text containing different types of entities that you will label. This is the list of entity types to label: Deity, Mythological_king, Cretaceous_dinosaur, Aquatic_mammal, Aquatic_animal, Goddess. Label the enities by surrounding them with tags like '<Cretaceous_dinosaur> Beipiaognathus </Cretaceous_dinosaur>'."""
        },
        clean_prompt, clean_response, error_prompt, error_response
    ]
```

The significant change to the baseline code to create the Gemini error-augmented prompt structure style is to `get_chat_history`. Instead of using a basic pair of input text and labeled text, each pair adds either spelling mistakes, strange capitalization, filler words, or other noise. This is intended to help the model learn to parse through the noise and still correctly label each token with the proper tag. All other functions are the same as in the baseline implementation.

```python
# A package to generate random sentences: https://pypi.org/project/wonderwords/
from wonderwords import RandomSentence

# This function creates the chat history using the error-augmented prompts structure. The idea is to inject errors, misspellings, and filler words into the prompts in the hopes that the model will still learn to pick up the correct context in order to label each token correctly.
def get_chat_history(shots, dataset, entity_types_list, convert_bio_to_prompt_fn):
    original_text = "Once paired in later myths with her Titan brother Hyperion as her husband, mild-eyed Euryphaessa, the far-shining one of the Homeric Hymn to Helios, was said to be the mother of Helios (the Sun), Selene (the Moon), and Eos (the Dawn)."
    original_label = "Once paired in later myths with her Titan brother <Deity> Hyperion </Deity> as her husband, mild-eyed Euryphaessa, the far-shining one of the Homeric Hymn to
```

```
    Helios, was said to be the mother of Helios (the Sun), <Goddess> Selene </Goddess> (the Moon), and <Goddess> Eos </Goddess> (the Dawn)."

    # Error 1: Misspelling and Case Change. This message contains misspelled words and words with weird casing. The LLM should be able to handle these malformed messages in the same way as the basic ones.
    error_text_1 = original_text.replace("Hyperion", "Hipyrion").replace("Selene", "sELenE")
    err_label_1 = original_label.replace("Hyperion", "Hipyrion").replace("Selene", "sELenE")

    # Error 2: Word Substitution and Noise Words. This message substitutes some words and adds other irrelevants words, none of which should be tagged as a special entity.
    error_text_2 = original_text.replace("Titan brother", "large sibling").replace("far-shining", "bright and beautiful") + " In addition, a random sea monster also appeared."
    error_label_2 = original_label.replace("Titan brother", "large sibling").replace("far-shining", "bright and beautiful") + " In addition, a random sea monster also appeared."

    # Error 3: Irrelevant Entity. This message contains both noise words and an entity which doesn't fit under any of the tags we have specified.
    error_text_3 = "The is some text which doesn't do anything. " + original_text +  " Also, King Arthur was said to be a great leader."
    error_label_3 = "The is some text which doesn't do anything. " + original_label +  " Also, King Arthur was said to be a great leader."

    # Create separate prompts with original and error-augmented data
    clean_prompt = {'role': USER_STR, MSG_STR: f"""Text: {original_text}"""}
    clean_response = {'role': SYSTEM_STR, MSG_STR: f"""Labels: {original_label}"""}

    error_prompt_1 = {'role': USER_STR, MSG_STR: f"""Text: {error_text_1}"""}
    error_response_1 = {'role': SYSTEM_STR, MSG_STR: f"""Labels: {err_label_1}"""}

    error_prompt_2 = {'role': USER_STR, MSG_STR: f"""Text: {error_text_2}"""}
    error_response_2 = {'role': SYSTEM_STR, MSG_STR: f"""Labels: {error_label_2}"""}

    error_prompt_3 = {'role': USER_STR, MSG_STR: f"""Text: {error_text_3}"""}
    error_response_3 = {'role': SYSTEM_STR, MSG_STR: f"""Labels: {error_label_3}"""}

    # Join the starting message with instructions to the rest of the formed messages
    messages = [
        {'role': SYSTEM_STR, MSG_STR:
        """You will be given input text containing different types of entities that you will label. This is the list of entity types to label: Deity, Mythological_king, Cretaceous_dinosaur, Aquatic_mammal, Aquatic_animal, Goddess. Label the enities by surrounding them with tags like '<Cretaceous_dinosaur> Beipiaognathus </Cretaceous_dinosaur>'."""
        },
        clean_prompt, clean_response, error_prompt_1, error_response_1, error_prompt_2, error_response_2, error_prompt_3, error_response_3]

    s = RandomSentence()

    # For the rest of the prompts, we want to add filler words and sentences, hoping that the model will learn to handle/ignore these irrelevant sections without compromising its ability to identify the tokens which need to be specially labelled.
    for i in range(shots):
        example = dataset[i]
        # Create a random (but grammatically correct) sentence
        sent = s.sentence()
        # To introduce a bit of extra pseudo-randomness, alternate if the filler sentence is at the beginning or the end of the proper message
        if i % 2 == 0:
            user_message = {
```

```
                'role': USER_STR, MSG_STR: """Text: {} {}""".format(sent, get_message(example))
            }
            system_message = {
                'role': SYSTEM_STR, MSG_STR: """Labels: {} {}""".format(sent, convert_bio_to_prompt_fn(example))
            }
        else:
            user_message = {
                'role': USER_STR, MSG_STR: """Text: {} {}""".format(get_message(example), sent)
            }
            system_message = {
                'role': SYSTEM_STR, MSG_STR: """Labels: {} {}""".format(convert_bio_to_prompt_fn(example), sent)
            }
        # Append all of the messages as normal
        messages.append(user_message)
        messages.append(system_message)

    return messages
```

Section A4: Experimental results code

```
# The eval code is the same as the baseline implementation
def run_eval(dataset, shots, backend):

  for example in tqdm(dataset, total=len(dataset), desc="Evaluating", position=tqdm._get_free_pos()):

      # String list of labels (BIO)
      true_labels = [labels_int2str[l] for l in example['ner_tags']]
      example_tokens = example['tokens']

      response_text = call_api_openai(shots, example) if backend == "openai" else call_api_cohere(shots, example)

      # String list of predicted labels (BIO)
      predictions, generated_tokens = convert_response_to_bio(response_text)

      if len(example['ner_strings']) != len(predictions):
            predictions = example['ner_strings']

      # Handle case where the generated text doesn't align with the input text.
      # Basically, we'll eval everything up to where the two strings start to diverge.
      # We relax this slightly by ignoring punctuation (sometimes we lose a paren or something,
      # but that's not catastrophic for eval/tokenization).
      # Just predict 'O' for anything following mismatch.
      matching_elements = [strip_punct(i) == strip_punct(j) for i, j in zip(example_tokens, generated_tokens)]

      if False in matching_elements:
         last_matching_idx = matching_elements.index(False)
      else:
         last_matching_idx = min(len(generated_tokens), len(example_tokens))

      predictions = predictions[:last_matching_idx] + ['O']*(len(example_tokens)-last_matching_idx)
      metric.add(predictions=predictions, references=true_labels)

  return metric.compute()
```

```python
# Run the eval on the entire dev set
dev_examples_to_take = 0

dev_set = data_splits['dev']
if dev_examples_to_take > 0:
    dev_set = data_splits['dev'].select(range(dev_examples_to_take))

# The number of shots was changed according to the specific run
for num_shots in [5]:
    print(f"shots: {num_shots}")
    result = run_eval(dev_set, shots=num_shots, backend='openai')
    print(result)

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

# The rest of the code is written to create the plots for the experimental results
# The following two sections of code create the graphs to display the performance changes as a result of changing the prompt structure.
# These numbers were extracted directly from the output after the evaluation was run. This plot and the plot below display the same data, but with different x-axis values. This plot has the performance metrics on the x-axis.
labels = ["Accuracy", "Precision", "Recall", "F1 Score"]
perplexity = [0.964, 0.363, 0.242, 0.298]
gpt = [0.956, 0.370, 0.256, 0.302]
gemini = [0.958, 0.347, 0.263, 0.290]

plt.plot(labels, perplexity, "-o", label="Perplexity AI")
plt.plot(labels, gpt, "-o", label="ChatGPT")
plt.plot(labels, gemini, "-o", label="Gemini")
plt.legend(loc="best")
plt.title('Performance comparision between Perplexity AI, Gemini, and GPT prompts')
plt.xlabel('Performance metric')
plt.ylabel('Score')
plt.show()

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

# These numbers were extracted directly from the output after the evaluation was run. This plot has the run type on the x-axis.
acc = [0.964, 0.956, 0.958]
prec = [0.363, 0.370, 0.347]
rec = [0.242, 0.256, 0.263]
f = [0.298, 0.302, 0.290]
labels = ["Perplexity AI", "ChatGPT", "Gemini"]

plt.plot(labels, acc, "-o", label="Accuracy")
plt.plot(labels, prec, "-o", label="Precision")
plt.plot(labels, rec, "-o", label="Recall")
plt.plot(labels, f, "-o", label="F1")
plt.legend(loc="best")
plt.title('Performance comparision between Perplexity AI, Gemini, and GPT prompts')
plt.xlabel('Model which created prompts')
plt.ylabel('Score')
plt.show()

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

# The following code creates the graphs to display the performance changes as a result of increasing the number of shots
# These numbers were extracted directly from the output after the evaluation was run. This plot and the plot below display the same data, but with different x-axis values. This
```

```
    plot has the performance metrics on the x-axis.
zerosht = [0.965, 0.460, 0.239, 0.296]
onesht = [0.959, 0.322, 0.300, 0.311]
fivesht = [0.965, 0.389, 0.239, 0.296]
tensht = [0.965, 0.329, 0.212, 0.258]
twentysht = [0.948, 0.366, 0.258, 0.302]
fortysht = [0.946, 0.441, 0.306, 0.361]
labels = ["Accuracy", "Precision", "Recall", "F1 Score"]

plt.plot(labels, zerosht, "-o", label='0 Shot')
plt.plot(labels, onesht, "-o", label='1 Shot')
plt.plot(labels, fivesht, "-o", label='5 Shot')
plt.plot(labels, tensht, "-o", label='10 Shot')
plt.plot(labels, twentysht, "-o", label='20 Shot')
plt.plot(labels, fortysht, "-o", label='40 Shot')
plt.legend(loc="best")
plt.title('Performance changes as a function of number of few-shot examples')
plt.xlabel('Performance metric')
plt.ylabel('Score')
# for e in [onesht, fivesht, tensht, twentysht, fortysht]:
#     for x, y in zip(labels, e):
#         plt.annotate(text=str(y), xy=(x, y))
plt.show()

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

# These numbers were extracted directly from the output after the evaluation was run. This
    plot has the number of shots on the x-axis.
accuracy = [0.965, 0.959, 0.965, 0.965, 0.948, 0.946]
precision = [0.460, 0.322, 0.389, 0.329, 0.366, 0.441]
recall = [0.239, 0.300, 0.239, 0.212, 0.258, 0.306]
f1 = [0.296, 0.311, 0.296, 0.258, 0.302, 0.361]
labels = ["0 shot", "1 shot", "5 shots", "10 shots", "20 shots", "40 shots"]

plt.plot(labels, accuracy, "-o", label="Accuracy")
plt.plot(labels, precision, "-o", label="Precision")
plt.plot(labels, recall, "-o", label="Recall")
plt.plot(labels, f1, "-o", label="F1 Score")
plt.legend(loc="best")
plt.title('Performance changes as a function of number of few-shot examples')
plt.xlabel('Number of shots')
plt.ylabel('Score')
plt.show()
```

Section A5: Analysis and Discussion code

Code used to generate Figures 10 - 12

```
# Code used to create the graphs of tag distribution across the dev and training sets. This
    code was used in the analysis of how the distribution of tags in the datasets affected
    the performance of different prompting styles
import json
import re
import matplotlib.pyplot as plt
from collections import Counter

tags = Counter()
tags_train = Counter()
tags_dev = Counter()

# Go through the dev and training datasets and collect the ner strings for each sample
for entry in data_splits['dev']:
```

```
    tags.update(entry['ner_strings'])
    tags_dev.update(entry['ner_strings'])
for entry in data_splits['train']:
    tags.update(entry['ner_strings'])
    tags_train.update(entry['ner_strings'])

total = sum(tags.values())

# Sort by the name of the tag, ignoring the B- and I-
def custom_sort(tag):
    return tag[2:]

def plot_each(cnt, st):

    # Extract all labels that aren't 'O'
    nonO = Counter({k: c for k, c in cnt.items() if k != "O"})

    # The x-axis is the tags sorted in alphabetical order
    labels = list(nonO.keys())
    sorted_labels = sorted(labels, key=custom_sort)
    sorted_counts = [nonO[label] for label in sorted_labels]

    plt.bar(sorted_labels, sorted_counts)
    plt.xlabel('NER labels')
    plt.xticks(rotation=90)
    plt.ylabel('Occurrences')
    if st == "all":
        plt.title('Occurrences of all NER labels without "O" across all datasets')
    else:
        plt.title('Occurrences of all NER labels without "O" in the ' + st + ' set')

    plt.show()

plot_each(tags_train, "training")
plot_each(tags_dev, "dev")
plot_each(tags, "all")
```

Code used to create graphs seen in Figures 13 - 18

```
# A function to create the graphs for each tag. These graphs were used in the analysis of performance of different prompting styles.
def plot_by_tag(tag, tag_prec, tag_rec, tag_f1, forty, perp, chatpt, gem):
    # These labels are the types of runs, the first graph uses them on the x-axis. The y-axis is the performance metric score scale.
    labels = ["40 shot", "Perplexity AI", "ChatGPT", "Gemini"]
    plt.plot(labels, tag_prec, "-o", label="Precision")
    plt.plot(labels, tag_rec, "-o", label="Recall")
    plt.plot(labels, tag_f1, "-o", label="F1 Score")
    plt.legend(loc="best")
    plt.title('Performance on the "' + tag + '" tag with modified prompts')
    plt.xlabel('Run type')
    plt.ylabel('Score')
    plt.show()

    # These labels are the performance metrics, the second graph uses them on the x-axis. The y-axis is the performance metric scores.
    labels = ["Precision", "Recall", "F1"]
    plt.plot(labels, forty, "-o", label="40 shot")
    plt.plot(labels, perp, "-o", label="Perplexity AI")
    plt.plot(labels, chatpt, "-o", label="ChatGPT")
    plt.plot(labels, gem, "-o", label="Gemini")
    plt.legend(loc="best")
```

```
    plt.title('Performance on the "' + tag + '" tag with modified prompts')
    plt.xlabel('Metric')
    plt.ylabel('Score')
    plt.show()
```

References

Bsharat S. M., Myrzakhan A., Shen Z., 2024, Principled Instructions Are All You Need for Questioning LLaMA-1/2, GPT-3.5/4 (arXiv:2312.16171)

Chen B., Zhang Z., Langrené N., Zhu S., 2023, Unleashing the potential of prompt engineering in Large Language Models: a comprehensive review (arXiv:2310.14735)

Eliot L., 2023, Latest Prompt Engineering Technique Inventively Transforms Imperfect Prompts into Superb Interactions for Using Generative AI, https://www.forbes.com/sites/lanceeliot/2023/07/26/latest-prompt-engineering-technique/-inventively-transforms-imperfect/-prompts-into-superb-interactions-for/-using-generative-ai/?sh=5af16bdb2c31

Liu P., Yuan W., Fu J., Jiang Z., Hayashi H., Neubig G., 2021, Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing (arXiv:2107.13586)

Reynolds L., McDonell K., 2021, Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm (arXiv:2102.07350)

Tang L., Peng Y., Wang Y., Ding Y., Durrett G., Rousseau J. F., 2023, Less Likely Brainstorming: Using Language Models to Generate Alternative Hypotheses (arXiv:2305.19339)

The New Stack 2023, Prompt Engineering: Get LLMs to Generate the Content You Want, https://thenewstack.io/prompt-engineering-get-llms-to/-generate-the-content-you-want/

Weird_Foundation5044 2023, Is it just me or is Prompt Engineering basically..., https://www.reddit.com/r/LocalLLaMA/comments/182pjyg/is_it_just_me_or_is_prompt_engineering_basically/

Weng L., 2023, lilianweng.github.io