Name: OLATUNJI Damilare Emmanuel **AndrewID:** dolatunj

Title:

Enhancing Entity Recognition for Dinosaur and Deity Naming Patterns Using BERT Fine-Tuning and LLM Prompt Engineering

Introduction:

The intersection of paleontology and sociolinguistics provides a unique lens through which to explore the etymology of dinosaur names, many of which appear inspired by mythology, particularly deities [1]. Traditional methods for uncovering these naming conventions involve manual annotation of large corpora—a tedious process that delays deeper cultural analysis. By leveraging natural language processing (NLP) and machine learning, this study focuses on exploring two modern approaches to sequence labelling: (1) fine-tuning a pretrained BERT model and (2) utilizing in-context learning with a large language model (LLM). This study not only evaluates their performance but also investigates the role of creative prompt engineering in optimizing LLM outputs. These methods promise to accelerate the annotation process, providing insights into cultural patterns embedded in dinosaur naming conventions.

Methodology:

This section outlines the methodology to the two complementary approaches to the sequence labelling task of identifying entities in text. (1) Fine-Tuning BERT, (2) LLM prompting for entity labeling.

1. **Fine-tuning BERT**

   a. **Data Pre-processing:** The datasets for training, validation, and testing are loaded and prepared using Hugging Face's datasets library. Each token in the text is annotated with BIO-encoded labels, representing entities like "Deity" or "Cretaceous_dinosaur." To ensure compatibility with the BERT tokenizer, entity labels are mapped to integer indices. The bert-base-cased tokenizer is employed to tokenize the text, splitting words into subword units where necessary. A custom alignment function is implemented to map token-level labels to the subword-level tokens, ensuring accurate label propagation across all datasets.

   b. **Model Development:** The BERT architecture is extended with a classification head to predict labels for each token. The classification layer consists of a fully connected layer applied to the contextual embeddings produced by BERT's final hidden layer. The model is fine-tuned on the labeled training dataset using the AdamW optimizer, with hyperparameters such as learning rate, batch size, and dropout probability optimized for performance. A data collator handles batch processing, ensuring that the input sequences and labels are efficiently aligned during training. To enhance generalization, the model's performance is validated on a separate development set after each training epoch.

   c. **Evaluation:** The evaluation phase uses the **seqeval** library to compute span-level precision, recall, and F1-score, assessing the model's ability to identify entity boundaries and types. Final testing is performed on an unseen dataset to measure the generalization capability of the trained model. The predictions are saved in JSON format for further analysis. Key metrics, including accuracy and F1-score, are reported to provide a comprehensive assessment of the model's performance.

2. **LLM prompting for entity labeling:**
   Under this section, the processes adopted involves, evaluating the impact of prompt engineering techniques (few-shot, zero-shot, retrieving and chain of thought and dynamic prompt construction) on Open AI ChatGPT 3.5-turbo model. Systematically varied prompt structures, and the number of demonstration examples (shots).

   a. **Implementation Details**
   The experiments were conducted using the OpenAI key. Below are some custom functions implemented to facilitate the experiments:

      [i] convert_bio_to_prompt: Converts BIO-encoded labels into an HTML-style tagged format suitable for inclusion in LLM prompts.
      [ii] convert_response_to_bio: Maps LLM predictions back to the BIO format for evaluation.
      [iii] get_chat_history and get_message: Constructs prompts by integrating task instructions, input text, and demonstration examples.

   Evaluation metrics, including precision, recall, and F1 score, were computed using the seqeval library.

   b. **Experiment variables: The number of shots (0, 1, 5, 10, 20, 30, 40, 50, 100).**
   c. **Prompt Construction**

Prompts were designed in an HTML-style format, where entity spans were enclosed in explicit tags (e.g., `<Cretaceous_dinosaur>Tyrannosaurus rex</Cretaceous_dinosaur>`). Each prompt included a mix of entity types (e.g., "Cretaceous_dinosaur," "Deity") to improve generalization.

d. Prompt Methods

Few-shot demonstration: The decision to use HTML-style tags for labeled spans was informed by [2] and [3] study, which demonstrated that explicit and well-structured prompts improve LLM performance on sequence labeling tasks. Additionally, few-shot learning has been shown to benefit from carefully curated, diverse examples [4]. This approach aims to maximize the alignment between the prompt format and the LLM's pretraining objectives, leveraging its inherent capabilities for text annotation and classification.

The constructed prompt in Figure 1 benefits from explicit labeling instructions combined with clear examples to guide the model in understanding the relationship between tokens and their entity types. By framing labeled entities with HTML-style tags, the prompt structure ensures unambiguous identification of spans while maintaining readability for the model. This design reflects best practices from [5] and [6] and provides the clarity needed to effectively leverage the model's contextual reasoning abilities during few-shot learning.

```
# 1. Few shots...
messages = [
    {'role': 'system', 'content':
     """Label the following text with the given entity types: Deity, Mythological_king, Cretaceous_dinosaur, Aquatic_mammal, Aquatic_animal,
     Use tags like '<Cretaceous_dinosaur> Beipiaognathus </Cretaceous_dinosaur>'."""
     },
     {'role': 'user', 'content': """Text: Once paired in later myths with her Titan brother Hyperion as her husband, mild-eyed Euryphaessa,
     {'role': 'system', 'content': """Labels: Once paired in later myths with her Titan brother <Deity> Hyperion </Deity> as her husband, mi
     {'role': 'user', 'content': """Text: From her ideological conception, Taweret was closely grouped with (and is often indistinguishable
]
```

Figure 1: Few-shot prompt template.

Chain-of-Thought (CoT) Templates: Recent studies demonstrated the ability of chain-of-thought (CoT) prompting to enhance reasoning and performance in large language models. As shown by [7] and [8], CoT prompting allows models to decompose complex tasks into intermediate reasoning steps. By incorporating explicit reasoning steps, CoT was combined with few-shot prompting technique, the model gains a clearer understanding of the relationships between tokens and their corresponding entity types. Figure 2 reflects the structured explanations accompanying the output. By breaking down decisions into logical steps, the model gains a clearer understanding of the relationships between tokens and their respective entity types [9]. The inclusion of reasoning in the prompt ensures that the model maintains consistency in predictions while enhancing its interpretability for sequence labeling tasks.

```
# 2. Chain of thoughts
messages = [
    {'role': 'system', 'content':
     """Label the following text with the given entity types: Deity, Mythological_king, Cretaceous_dinosaur, Aquatic_mammal, Aquatic
     Use tags like '<Cretaceous_dinosaur> Beipiaognathus </Cretaceous_dinosaur>'. For each label, explain the reasoning step by step

    {'role': 'user', 'content': """Text: Once paired in later myths with her Titan brother Hyperion as her husband, mild-eyed Euryph

    {'role': 'system', 'content':
     """Reasoning:
     - "Hyperion" is described as a Titan, which is a mythological figure, making it a <Deity>.
     - "Euryphaessa" is described as the mother of Helios (the Sun), aligning with mythological roles, making her a <Goddess>.
     - "Helios" represents the Sun in mythology, so it is labeled as <Deity>.
     - "Selene" is described as the Moon and associated with mythological figures, making her a <Goddess>.
     - "Eos" is described as the Dawn, also aligning with a mythological role, making her a <Goddess>.

     Labels: Once paired in later myths with her Titan brother <Deity> Hyperion </Deity> as her husband, mild-eyed <Goddess> Eurypha

    {'role': 'user', 'content':
     """Text: From her ideological conception, Taweret was closely grouped with (and is often indistinguishable from) several other
]
```

Figure 2A: Chain-of-Thought (CoT) Templates

Dynamic prompt construction template: Dynamic prompt construction was employed to adapt prompts based on the specific characteristics of the input text as it can be built on with few-shots, ensuring relevance and improved performance for entity recognition tasks [10]. By selecting examples that closely align with the input in terms of entity types, structure, or complexity, this approach enhances the model's ability to generalize and make accurate predictions. Studies such as [11] and [12] demonstrate that dynamically tailored prompts improve task-specific performance by focusing on contextually relevant demonstrations while maintaining efficiency. This method aligns with the model's pretraining objectives, enabling it to handle diverse inputs with greater precision.

```
# 3. Dynamic prompt construction
messages = [
    {'role': 'system', 'content':
     """Label the following text with the given entity types: Deity, Mythological_king, Cretaceous_dinosaur, A
     Use tags like '<Cretaceous_dinosaur> Beipiaognathus </Cretaceous_dinosaur>'. Select examples that closely

    # Dynamically selected demonstration 1
    {'role': 'user', 'content': """Text: The well-known dinosaur Tyrannosaurus rex lived during the late Creta
    {'role': 'system', 'content': """Labels: The well-known dinosaur <Cretaceous_dinosaur> Tyrannosaurus rex <

    # Dynamically selected demonstration 2
    {'role': 'user', 'content': """Text: Neptune was considered the god of the sea in Roman mythology, often d
    {'role': 'system', 'content': """Labels: <Deity> Neptune </Deity> was considered the god of the sea in Roma

    # Dynamically selected demonstration 3
    {'role': 'user', 'content': """Text: Selene was frequently associated with the Moon and was part of Greek 
    {'role': 'system', 'content': """Labels: <Goddess> Selene </Goddess> was frequently associated with the Mo

    # Input text to label
    {'role': 'user', 'content': """Text: From her ideological conception, Taweret was closely grouped with (an
]
```

Figure 3A: Dynamic prompt construction template

Dynamic Prompt Implementation

```
import random

def get_chat_history(shots, dataset, entity_types_list, convert_bio_to_prompt_fn):
    """
    Generate a dynamic prompt with contextually relevant examples for the model.

    :param shots: Number of examples to include for few-shot learning.
    :param dataset: The dataset to sample examples from.
    :param entity_types_list: List of entity types (e.g., ['Deity', 'Cretaceous_dinosaur']).
    :param convert_bio_to_prompt_fn: Function to convert BIO examples to a string.
    :return: List of message dictionaries for the model's chat history.
    """
    chat_history = []

    # Add the system message introducing the task
    chat_history.append({
        'role': 'system',
        'content': f"""Label the following text with the given entity types: {', '.join(entity_types_list)}.
                    Use tags like '<Cretaceous_dinosaur> Beipiaognathus </Cretaceous_dinosaur>'.
                    Select examples that closely match the context or sentence structure of the input text."""
    })

    # Select contextually relevant examples dynamically
    selected_examples = []
    for example in dataset:
        if len(selected_examples) >= shots:
            break
        # Simple heuristic: check if input text shares entity types or structure with the example
        if any(entity in example['content'] for entity in entity_types_list):
            selected_examples.append(example)

    # If not enough examples are found, pad with random examples
    while len(selected_examples) < shots:
        selected_examples.append(random.choice(dataset))

    # Add the dynamically selected examples to the chat history
    for example in selected_examples:
        formatted_example = convert_bio_to_prompt_fn(example)
        chat_history.append({
            'role': 'user',
            'content': f"Text: {example['content']}"
        })
        chat_history.append({
            'role': 'system',
            'content': f"Labels: {formatted_example}"
        })

    return chat_history
```

Figure 3B: Dynamic prompt construction template

Chain-of-thought Implementation

```
# Chain-of-Thought Implementation

def get_chat_history(shots, dataset, entity_types_list, convert_bio_to_prompt_fn):
    """
    Generate a list of prompt examples for the model, using `shots` as the number of demonstration examples.

    :param shots: Number of examples to include for few-shot learning.
    :param dataset: The dataset to sample examples from.
    :param entity_types_list: List of entity types (e.g., ['Deity', 'Cretaceous_dinosaur']).
    :param convert_bio_to_prompt_fn: Function to convert BIO examples to a string.
    :return: List of message dictionaries for the model's chat history.
    """
    chat_history = []

    # First, add the system message that introduces the entity types and explains the chain of thought process
    chat_history.append({
        'role': 'system',
        'content': 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: {', '.join(entity_types_list)}.\
                    For each example, think step-by-step and provide detailed reasoning before labeling the entities.\
                    Label the entities by surrounding them with tags like '<Cretaceous_dinosaur> Beipiaognathus </Cretace
    })

    # Now, add the user messages based on the dataset examples
    for i in range(shots):
        example = dataset[i]
        print("Example:", example)
        formatted_example = convert_bio_to_prompt_fn(example)
        print("Example:", formatted_example)

        # Example 1: Show the text with expected labels and include reasoning steps
        chat_history.append({
            'role': 'user',
            'content': f"Text: {example['content']}\n"
                       f"Step 1: Identify the entities in the text.\n"
                       f"Step 2: Determine the type of each entity based on the context.\n"
                       f"Step 3: Label the entities by surrounding them with the appropriate tags.\n"
                       f"Labels: {formatted_example}"
        })

    return chat_history
✓ 0.0s
```

Figure 2B: Chain-of-Thought (CoT) Templates

Experimental Results

This section presents the result of fine-tunning with BERT and experimentation with various prompt engineering technique (few-shot, chain-of-thought, and dynamic prompting) to identify the best performing LLM strategy. Using accuracy, precision, recall, and F1-scores to present a clear comparison and highlight where the LLM excels or falls short relative to the baseline approach.

Finetune BERT Approach -

BERT as a fine-tuned model provides the result in Table 1 with a final accuracy of 0.953 and F1-score of 0.461.

S/N	Approach	Shots	Precision	Recall	F1	Accuracy
1.	Finetune BERT	-	0.4280	0.500	0.461	0.953

Table 1: Baseline model - BERT

Experimentation with prompt engineering:

The experiments with chain-of-thought, and dynamic prompts are aimed at improving upon or evaluating how close the OpenAI LLM approaches with the baseline approach (few-shot). Each prompt template (few-shot, chain-of-thought, and dynamic prompts) is designed to leverage the capabilities of the LLM.

Prompt engineering technique 1 - (Few-shot LLM Baseline):

Table 1 summarizes the performance of the few-shot LLM baseline approach across various number of shots (0, 1, 5, 10, 20, 30 and 40). This data serves as a benchmark for understanding the impact of few-shot prompting on entity recognition task.

S/N	Approach	Shots	Precision	Recall	F1	Accuracy
1.	Zero-shot LLM Baseline	0	0.2235	0.0576	0.0916	0.9665
2.	Few-shot LLM Baseline	1	0.2647	0.2455	0.2547	0.9580
3.	Few-shot LLM Baseline	5	0.2079	0.1758	0.1905	0.9559
4.	Few-shot LLM Baseline	10	0.1993	0.1727	0.1851	0.9568
5.	Few-shot LLM Baseline	20	0.1898	0.2030	0.1962	0.9468
6.	Few-shot LLM Baseline	30	0.1942	0.2030	0.1985	0.9446
7.	Few-shot LLM Baseline	40	0.2000	0.2030	0.2015	0.9350
Prompt approach: Few-shot - An effective technique to prompting where you provide exemplars (i.e., demonstrations).						

Table 1: **Few-shot prompt template.**

Figure 1A and 1B visualizes how the performance metrics (precision, recall and F1-score) changes as a function of the number of shots for the few-shot LLM baseline approach. Each plot uses the number of demonstration examples (shots) as the x-axis and evaluation metrics as the y-axis, illustrating trends in model performance. These plots allow us to assess the relationship between the number of examples provided to the model and its ability to predict entities accurately, aiding in the comparison of baseline and enhanced prompt engineering techniques.

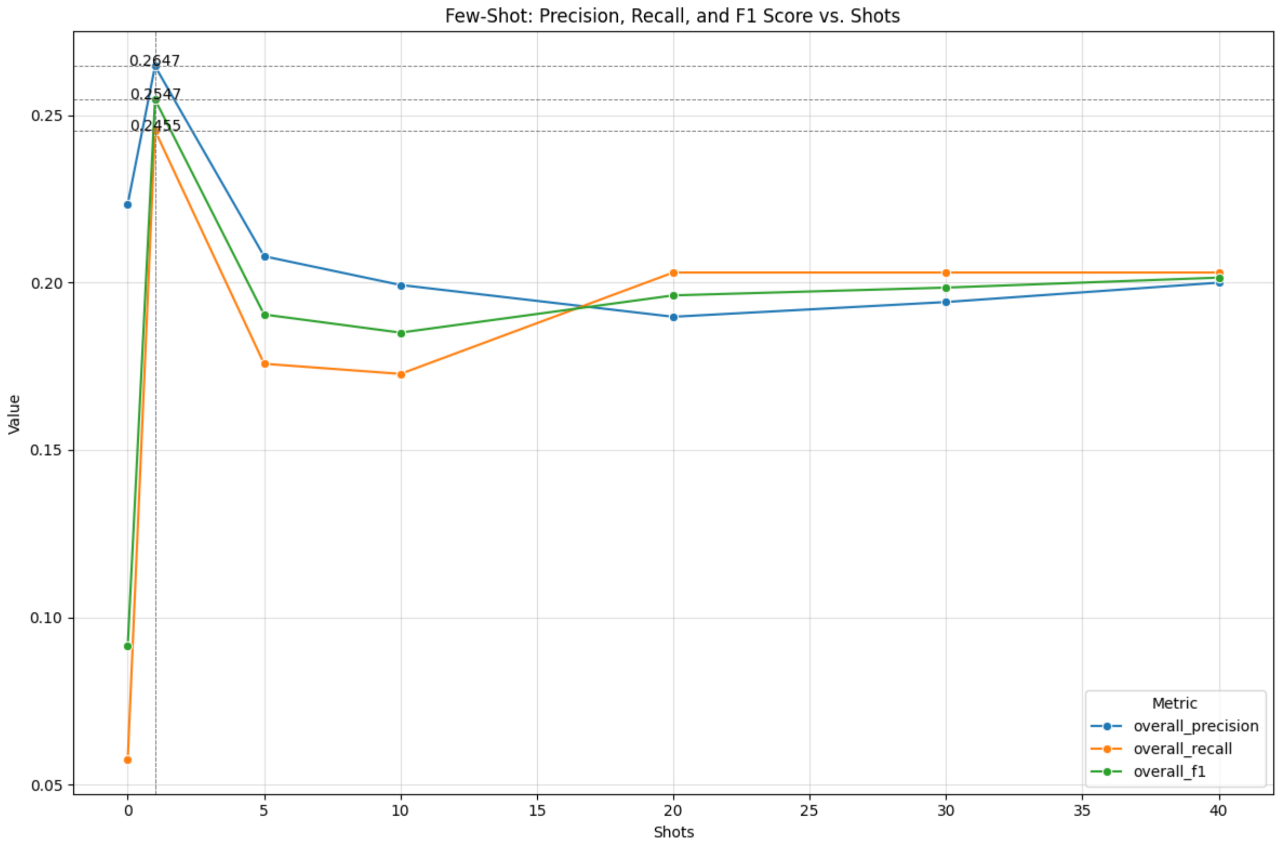

Figure 1A – Few-Shot: Precision, Recall, and F1 Score

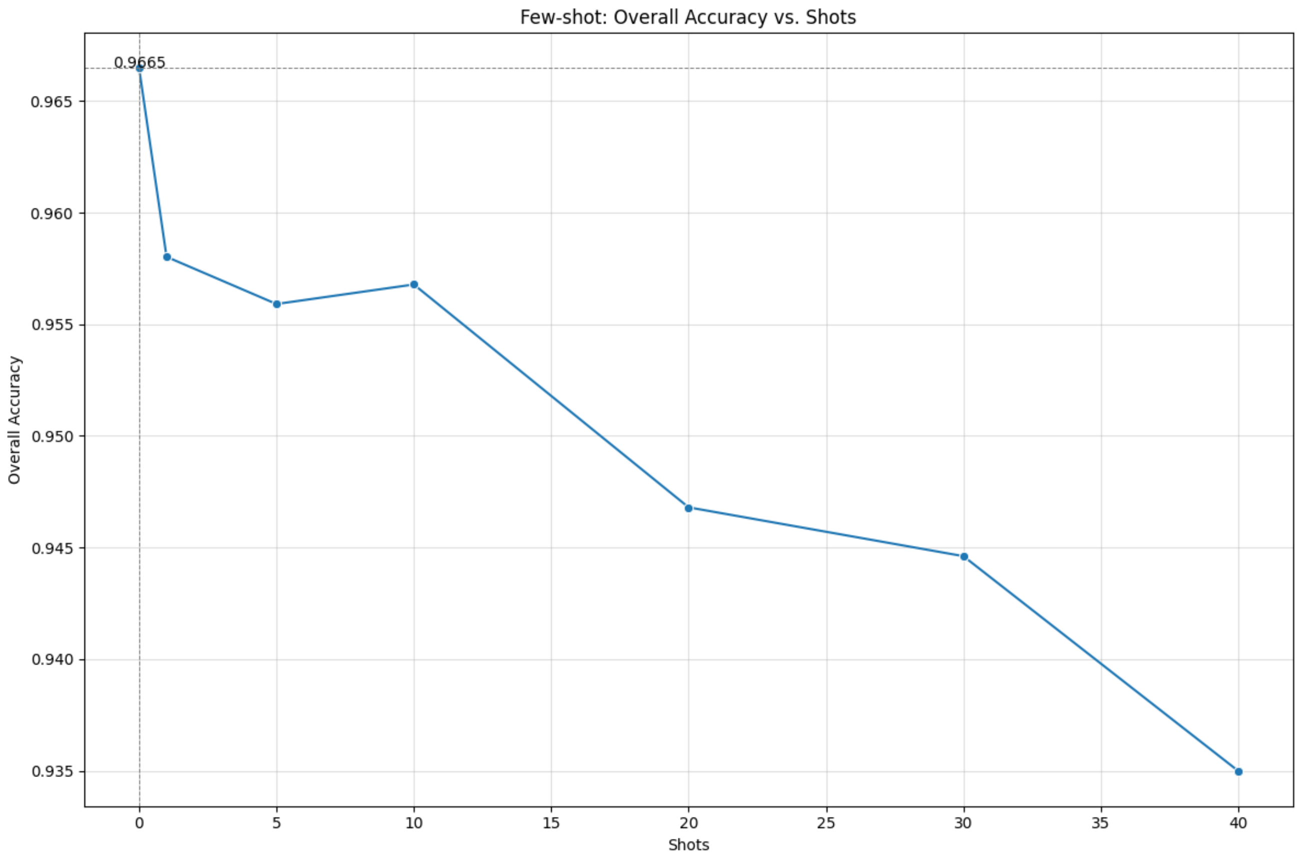

Figure 1B – Overall Accuracy

Prompt engineering technique 2 - (Chain-of-thought):

Table 2 summarizes the performance of chain-of-thought approach across various number of shots (0, 1, 5, 10, 20, 30 and 40). The table reports key evaluation metrics (precision, F1-score and accuracy).

S/N	Approach	Shots	Precision	Recall	F1	Accuracy
1.	Chain-of-thought	0	0.2424	0.0485	0.0808	0.9677
2.	Chain-of-thought	1	0.2441	0.2515	0.2478	0.9548
3.	Chain-of-thought	5	0.2096	0.1727	0.1894	0.9551
4.	Chain-of-thought	10	0.2116	0.1879	0.1990	0.9570
5.	Chain-of-thought	20	0.1994	0.2152	0.2070	0.9545
6.	Chain-of-thought	30	0.1860	0.1939	0.1899	0.9389
7.	Chain-of-thought	40	0.1737	0.1758	0.1747	0.9262
Prompt approach: Chain-of-thought template						

Table 2: **Chain-of-thought template.**

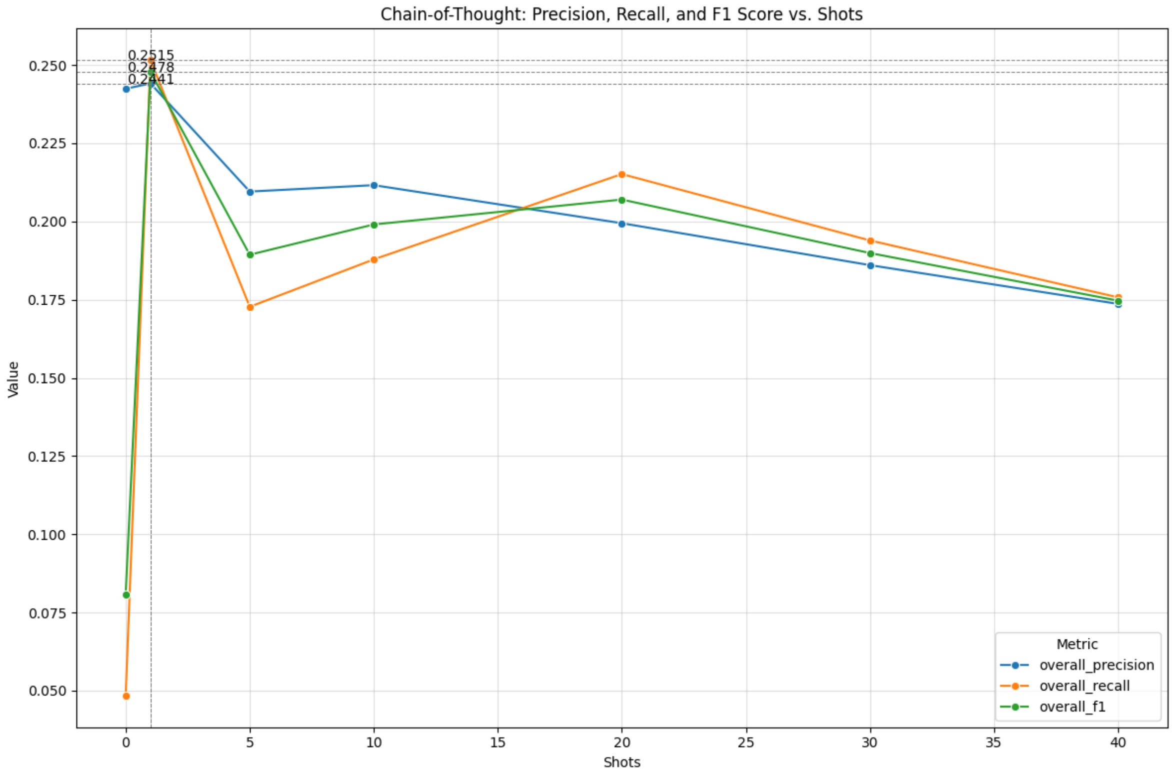

Table 2A: **Chain-of-Thought: Precision, Recall, and F1-Score.**

Figure 2A and 2B visualizes how the performance metrics (precision, recall and F1-score) changes as a function of the number of shots. Each plot uses the number of demonstration examples (shots) as the x-axis and evaluation metrics as the y-axis, illustrating trends in model performance. These plots allow us to assess the relationship between the number of examples provided to the model and its ability to predict entities accurately.

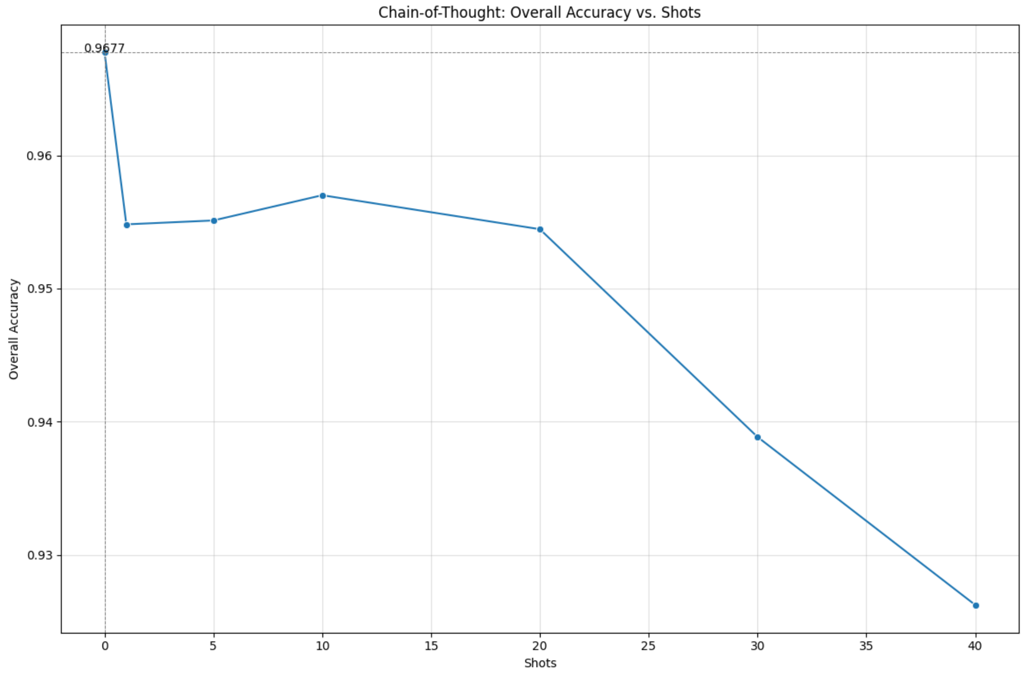

Table 2B: **Chain-of-Thought: Precision, Recall, and F1-Score.**

Prompt engineering technique 3: Dynamic prompting

Table 3 summarizes the performance of dynamic prompting construction across various number of shots (0, 1, 5, 10, 20, 30 and 40). The table reports key evaluation metrics (precision, F1-score and accuracy).

S/N	Approach	Shots	Precision	Recall	F1	Accuracy
1.	Dynamic prompting	0	0.2857	0.0424	0.0739	0.9687
2.	Dynamic prompting	1	0.2840	0.2909	0.2874	0.9572
3.	Dynamic prompting	5	0.2230	0.1939	0.2075	0.9567
4.	Dynamic prompting	10	0.2261	0.1939	0.2088	0.9573
5.	Dynamic prompting	20	0.1940	0.2152	0.2040	0.9529
6.	Dynamic prompting	30	0.1816	0.1909	0.1861	0.9359
7.	Dynamic prompting	40	0.1988	0.2000	0.1994	0.9283
Prompting approach: Dynamic prompt template						

Table 3**: Dynamic prompt template**

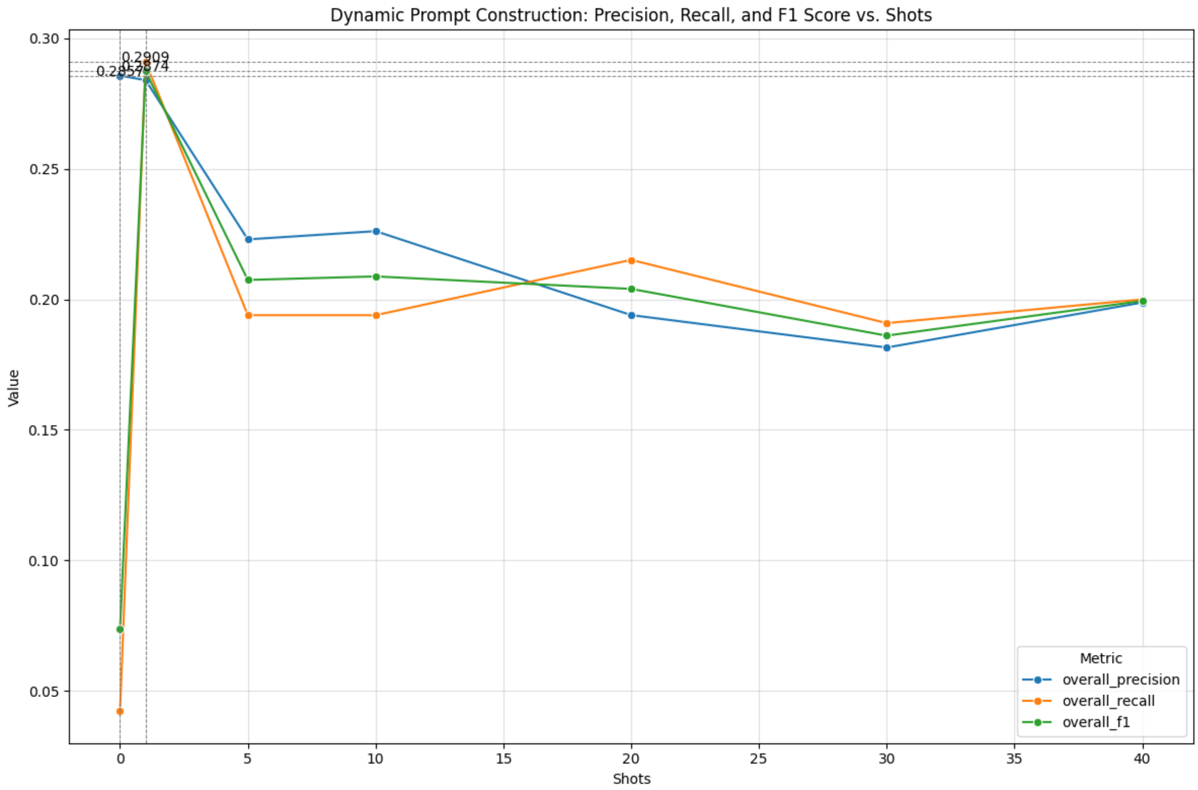

Table 3A: **Dynamic Prompt Construction: Precision, Recall, and F1-Score.**

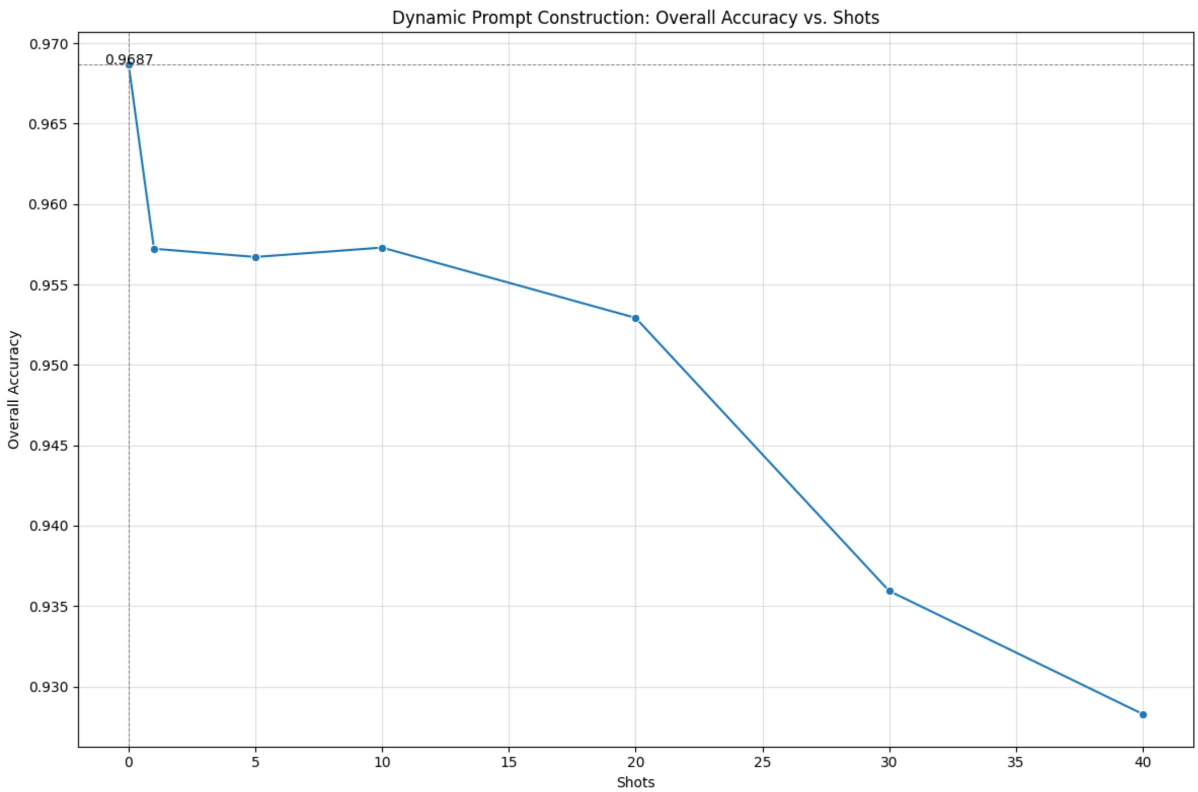

Table 3A: **Dynamic Prompt Construction: Overall Accuracy**

Analysis and Discussion

This section presents a detailed analysis of the performance of three prompt engineering approaches—Few-shot prompting, Chain-of-Thought prompting, and Dynamic Prompting—across multiple metrics: Precision, Recall, F1-score, and Accuracy. Visualized results highlight trends, strengths, and weaknesses, focusing on identifying optimal configurations for sequence labelling tasks.

1. **Performance Trends (Precision)**

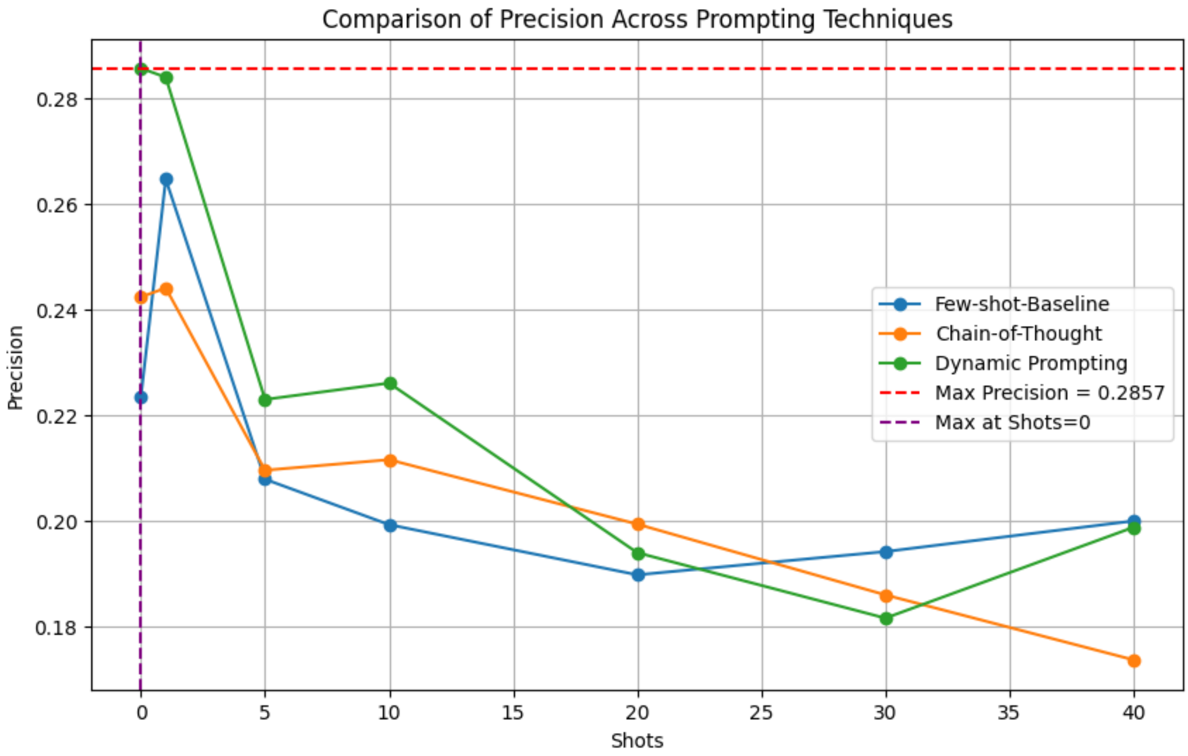

Figure 4A: Comparison of precision across prompting techniques

Dynamic Prompting demonstrated exceptional performance, achieving the highest precision of 0.2857 at 0 shots, as highlighted in Figure 4A. This indicates the effectiveness of dynamic prompts in adapting to input and enhancing initial precision. In comparison, Few-shot prompting started with a precision of 0.2235 at 0 shots, peaked at 0.2647 with 1 shot, and steadily declined as more examples were added, reflecting diminishing returns with increased complexity. Chain-of-Thought prompting, on the other hand, maintained stable precision across shots, showing a modest peak of 0.2441 at 1 shot and only minor fluctuations, thereafter, demonstrating its consistency and reliability over varying inputs.

2. **Performance Trends (Recall)**

Figure 4B shows that dynamic prompting achieved the highest recall of 0.2909 at 1 shot, showing its adaptability in capturing broader context with minimal examples. In contrast, Few-shot prompting plateaued at approximately 0.2030 after 20 shots, suggesting its limited capacity to capture additional context as more examples are added. Chain-of-Thought prompting displayed moderate and consistent recall across all shots, peaking at 0.2515 at 1 shot. This stability underscores its effectiveness in tasks requiring strong contextual reasoning and systematic understanding.

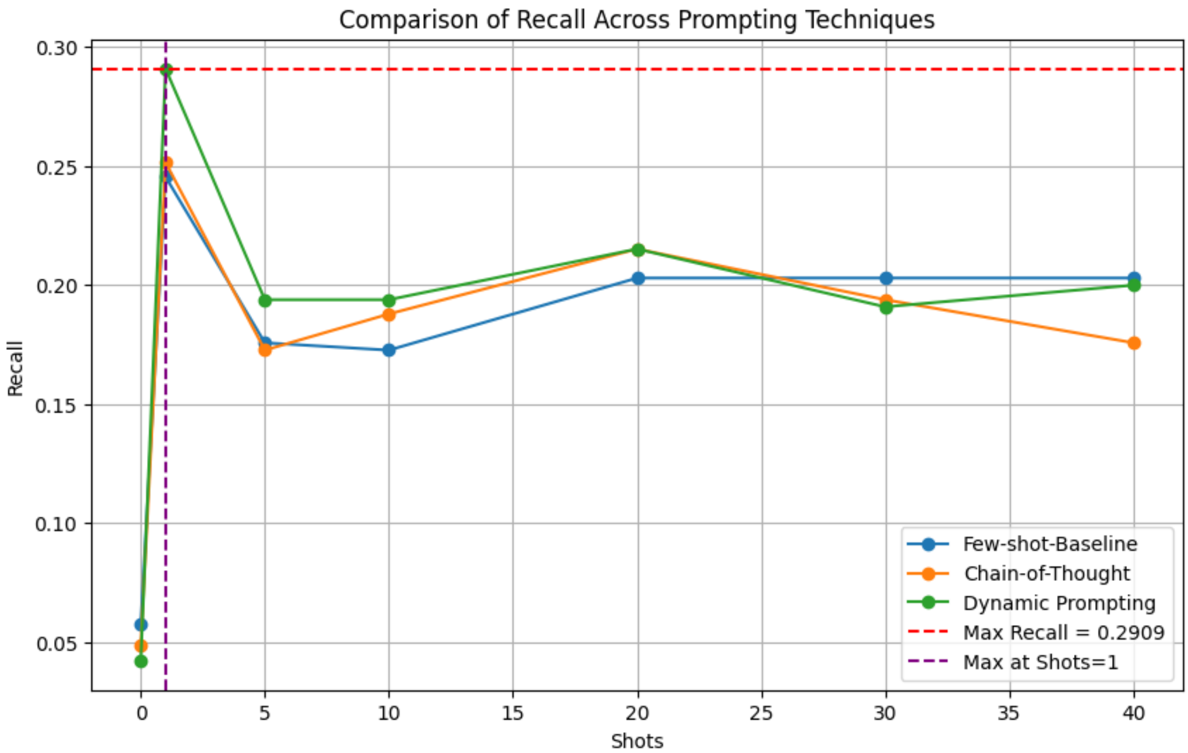

Figure 4B: Comparison of recall across prompting techniques

3. Performance Trends (F1-Score)

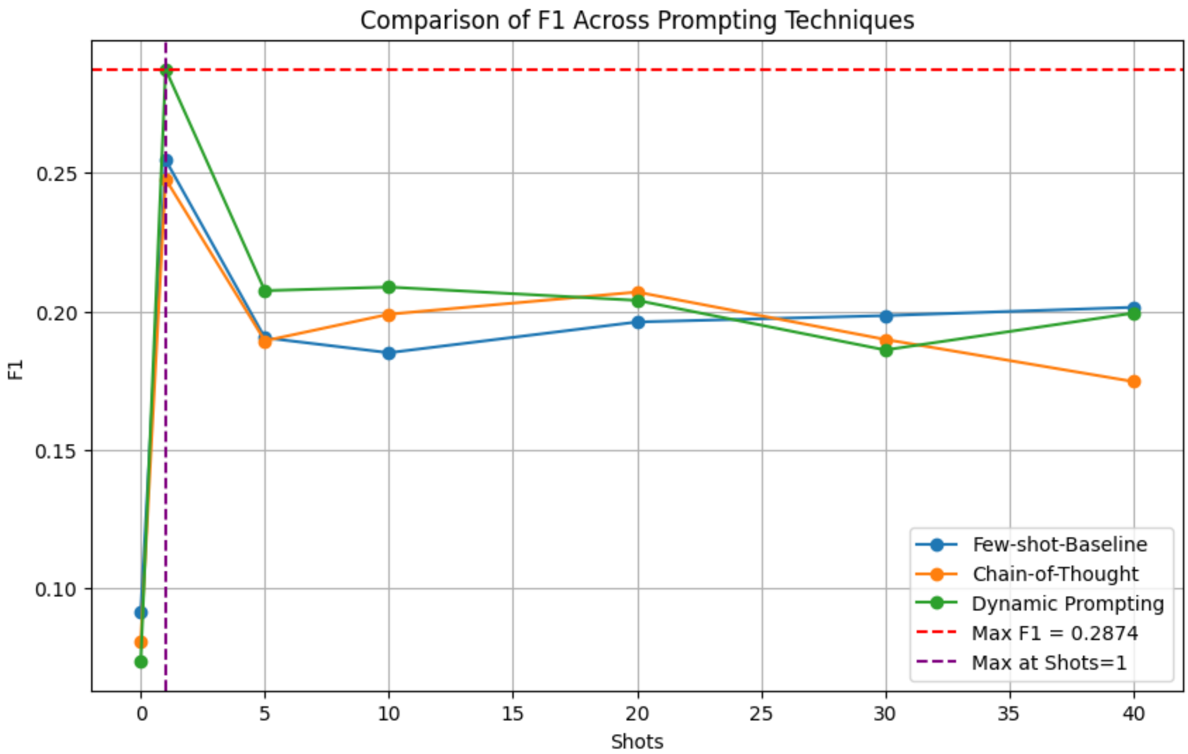

Figure 4C: Comparison of F1-Score across prompting techniques

- Dynamic Prompting Dominance: Dynamic prompting outperformed other methods in F1-score, achieving 0.2874 at 1 shot. This is significant, as F1 measures the balance between precision and recall.
- Few-shot Peak: Few-shot prompting reaches a peak F1-score of 0.2547 at 1 shot, followed by a decline. This indicates that Few-shot prompting is effective with minimal examples but struggles with scalability.
- While Chain-of-Thought prompting peaked at 0.2478 at 1 shot, it maintained a more consistent F1 trend across shots, suggesting stability in prediction quality.

4. Performance Trends (Accuracy)

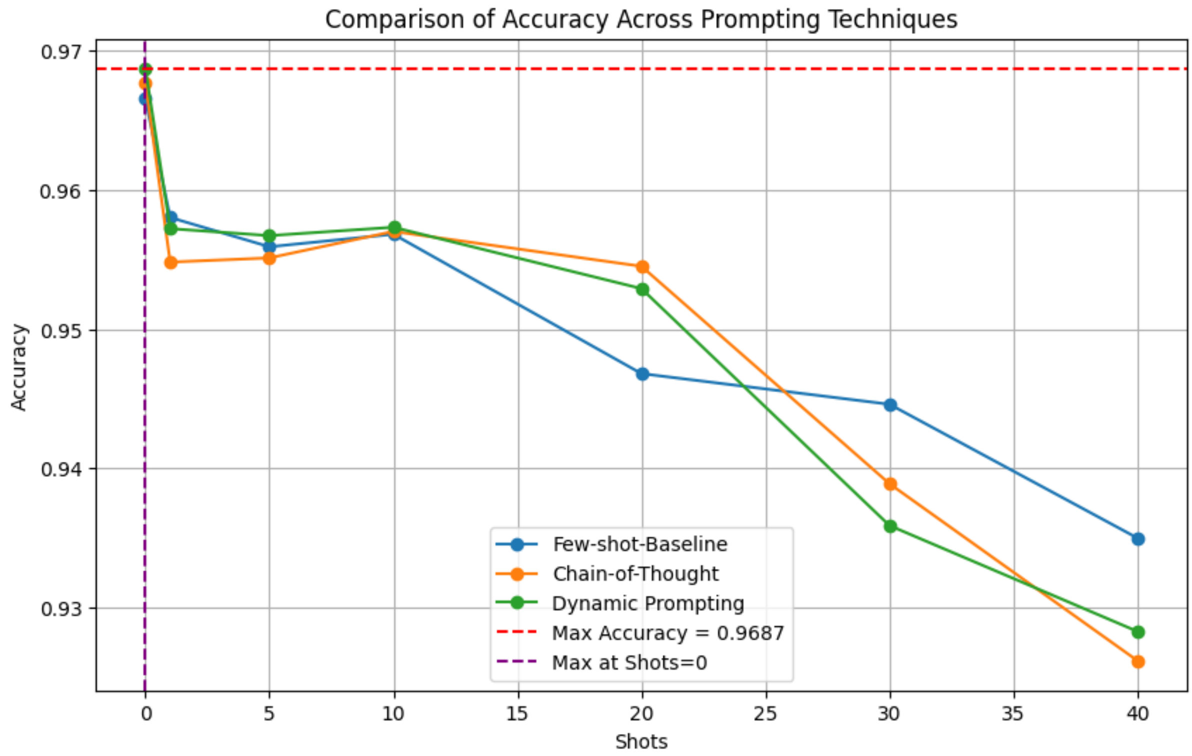

Figure 4D: Comparison of accuracy across prompting techniques

- Dynamic Prompting Lead: Dynamic prompting achieved the highest accuracy of 0.9687 at 0 shots, showing its initial robustness. However, accuracy declines with more shots, reflecting possible overfitting.
- Few-shot Decline: Few-shot prompting showed a gradual decline in accuracy from 0.9665 (0 shots) to 0.9350 (40 shots), demonstrating reduced generalization with more examples.
- Chain-of-Thought Balance: Chain-of-Thought prompting maintained competitive accuracy across shots, peaking at 0.9677 at 0 shots but declining steadily.

Observations

1. **Few-shot Prompting**:
   - Few-shot prompting performs well with minimal examples but struggles as the number of shots increases. The decline in precision, recall, and accuracy indicates diminishing returns and potential overfitting to larger example sets.
2. **Chain-of-Thought Prompting**:
   - Chain-of-Thought prompting excels in maintaining stable performance across metrics, particularly in recall and F1-score. This method is well-suited for tasks requiring consistent reasoning and context interpretation.

3. **Dynamic Prompting**:
   - Dynamic prompting consistently achieved the highest metrics across all categories with fewer shots. Its adaptability and contextual relevance make it the most effective approach for precision and F1-score.

Conclusion

The analysis of the three prompting strategies—Few-shot, Chain-of-Thought, and Dynamic Prompting—reveals valuable insights into their optimal use cases and trade-offs for sequence labeling tasks. Among these approaches, dynamic prompting emerges as the most effective strategy for tasks that prioritize precision and F1-score, especially when using 0-1 shots. Its adaptability and ability to extract meaningful patterns from minimal examples make it particularly well-suited for achieving high-quality outputs. On the other hand, dynamic Prompting, despite its exceptional performance with fewer examples, shows a gradual decline in accuracy as the number of shots increases, suggesting potential overfitting or reduced adaptability with larger prompts.

Overall, selecting the appropriate prompting strategy depends on the task's specific requirements. For high precision and F1-score with minimal examples, Dynamic Prompting is ideal. For tasks requiring logical consistency and stability, Chain-of-Thought prompting is a robust choice. Understanding these trade-offs allows for informed decisions when applying prompt engineering techniques to optimize the performance of large language models in sequence labeling tasks.

References

[1] E. Nyberg and D. Mortensen, "11-411/11-611 Homework Assignment 4: Entity Recognition," Language Technologies Institute, School of Computer Science, Carnegie Mellon University, class notes.

[2] T. B. Brown *et al.*, “Language Models are Few-Shot Learners,” Jul. 22, 2020, *arXiv*: arXiv:2005.14165. doi: 10.48550/arXiv.2005.14165.

[3] M. Hofer, A. Kormilitzin, P. Goldberg, and A. Nevado-Holgado, “Few-shot Learning for Named Entity Recognition in Medical Text,” Nov. 13, 2018, *arXiv*: arXiv:1811.05468. doi: 10.48550/arXiv.1811.05468.

[4] “Few-Shot Learning for Data-Scarce Problems | by Amit Yadav - Freedium.” Accessed: Dec. 11, 2024. [Online]. Available: https://freedium.cfd/https://medium.com/@amit25173/few-shot-learning-for-data-scarce-problems-0a63698edd56

[5] “Few-Shot Prompting – Nextra.” Accessed: Dec. 11, 2024. [Online]. Available: https://www.promptingguide.ai/techniques/fewshot

[6] S. Min *et al.*, “Rethinking the Role of Demonstrations: What Makes In-Context Learning Work?,” Oct. 20, 2022, *arXiv*: arXiv:2202.12837. doi: 10.48550/arXiv.2202.12837.

[7] J. Wei *et al.*, “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models,” Jan. 10, 2023, *arXiv*: arXiv:2201.11903. doi: 10.48550/arXiv.2201.11903.

[8] “Chain-of-Thought Prompting – Nextra.” Accessed: Dec. 11, 2024. [Online]. Available: https://www.promptingguide.ai/techniques/cot

[9] “What is chain-of-thought prompting? Examples and benefits,” Search Enterprise AI. Accessed: Dec. 11, 2024. [Online]. Available: https://www.techtarget.com/searchenterpriseai/definition/chain-of-thought-prompting

[10] “Leveraging dynamic few-shot prompt with Azure OpenAI,” TECHCOMMUNITY.MICROSOFT.COM. Accessed: Dec. 13, 2024. [Online]. Available: https://techcommunity.microsoft.com/blog/fasttrackforazureblog/leveraging-dynamic-few-shot-prompt-with-azure-openai/4225235

[11] K. R, “Think Beyond Size: Dynamic Prompting for More Effective Reasoning,” Oct. 10, 2024, *arXiv*: arXiv:2410.08130. doi: 10.48550/arXiv.2410.08130.

[12] Avaamo, “Introducing Dynamic Prompts: Unlocking Magic Without the Dark Arts,” Medium. Accessed: Dec. 11, 2024. [Online]. Available: https://avaamo.medium.com/dynamic-prompts-unlocking-magic-without-the-dark-arts-ef56b7e852bc

Appendix

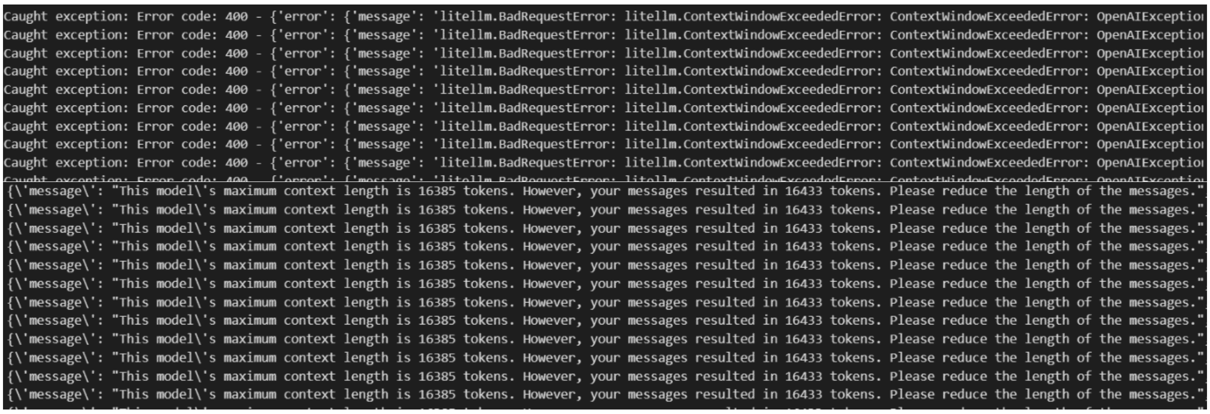
```
Caught exception: Error code: 400 - {'error': {'message': 'litellm.BadRequestError: litellm.ContextWindowExceededError: ContextWindowExceededError: OpenAIExceptio
Caught exception: Error code: 400 - {'error': {'message': 'litellm.BadRequestError: litellm.ContextWindowExceededError: ContextWindowExceededError: OpenAIExceptio
Caught exception: Error code: 400 - {'error': {'message': 'litellm.BadRequestError: litellm.ContextWindowExceededError: ContextWindowExceededError: OpenAIExceptio
Caught exception: Error code: 400 - {'error': {'message': 'litellm.BadRequestError: litellm.ContextWindowExceededError: ContextWindowExceededError: OpenAIExceptio
Caught exception: Error code: 400 - {'error': {'message': 'litellm.BadRequestError: litellm.ContextWindowExceededError: ContextWindowExceededError: OpenAIExceptio
Caught exception: Error code: 400 - {'error': {'message': 'litellm.BadRequestError: litellm.ContextWindowExceededError: ContextWindowExceededError: OpenAIExceptio
Caught exception: Error code: 400 - {'error': {'message': 'litellm.BadRequestError: litellm.ContextWindowExceededError: ContextWindowExceededError: OpenAIExceptio
Caught exception: Error code: 400 - {'error': {'message': 'litellm.BadRequestError: litellm.ContextWindowExceededError: ContextWindowExceededError: OpenAIExceptio
Caught exception: Error code: 400 - {'error': {'message': 'litellm.BadRequestError: litellm.ContextWindowExceededError: ContextWindowExceededError: OpenAIExceptio
{\'message\': "This model\'s maximum context length is 16385 tokens. However, your messages resulted in 16433 tokens. Please reduce the length of the messages."
{\'message\': "This model\'s maximum context length is 16385 tokens. However, your messages resulted in 16433 tokens. Please reduce the length of the messages."
{\'message\': "This model\'s maximum context length is 16385 tokens. However, your messages resulted in 16433 tokens. Please reduce the length of the messages."
{\'message\': "This model\'s maximum context length is 16385 tokens. However, your messages resulted in 16433 tokens. Please reduce the length of the messages."
{\'message\': "This model\'s maximum context length is 16385 tokens. However, your messages resulted in 16433 tokens. Please reduce the length of the messages."
{\'message\': "This model\'s maximum context length is 16385 tokens. However, your messages resulted in 16433 tokens. Please reduce the length of the messages."
{\'message\': "This model\'s maximum context length is 16385 tokens. However, your messages resulted in 16433 tokens. Please reduce the length of the messages."
{\'message\': "This model\'s maximum context length is 16385 tokens. However, your messages resulted in 16433 tokens. Please reduce the length of the messages."
{\'message\': "This model\'s maximum context length is 16385 tokens. However, your messages resulted in 16433 tokens. Please reduce the length of the messages."
{\'message\': "This model\'s maximum context length is 16385 tokens. However, your messages resulted in 16433 tokens. Please reduce the length of the messages."
{\'message\': "This model\'s maximum context length is 16385 tokens. However, your messages resulted in 16433 tokens. Please reduce the length of the messages."
{\'message\': "This model\'s maximum context length is 16385 tokens. However, your messages resulted in 16433 tokens. Please reduce the length of the messages."
```

Figure 3: Exceed maximum context length (No. of shots = 50)

BERT

```python
# Load the dataset
from datasets import ClassLabel, Sequence, load_dataset

# Load the dataset from JSON files for train, dev, and test splits
data_splits = load_dataset('json', data_files={'train': 'dinos_and_deities_train_bio.jsonl', 'dev':
'dinos_and_deities_dev_bio_sm.jsonl', 'test': 'dinos_and_deities_test_bio_nolabels.jsonl'})

# Define the file name containing the label names
label_names_fname = "dinos_and_deities_train_bio.jsonl.labels"

# Initialize a list to store the label names
labels_int2str = []
# Read the label names from the file and split them into a list
with open(label_names_fname) as f:
    labels_int2str = f.read().split()

# Print the label names
print(f"Labels: {labels_int2str}")

# Create a dictionary to map label names to their corresponding integer indices
labels_str2int = {l: i for i, l in enumerate(labels_int2str)}

# Cast the "ner_tags" column to a sequence of ClassLabel with the defined label names
data_splits.cast_column("ner_tags", Sequence(ClassLabel(names=labels_int2str)))

# Print the dataset splits to verify the changes
print(data_splits)
```

```python
# This dataset is split into a train, validation and test set, and each token has a label.
# Data from the dataset can generally be accessed like a Python dict.
print(data_splits['train'].features)

# Print the original sentence (which is whitespace tokenized).
example_input_tokens = data_splits['train'][8]['tokens']
print(f"Original tokens: {example_input_tokens}")

# Print the labels of the sentence.
example_ner_labels = data_splits['train'][8]['ner_tags']
print(f"NER labels: {example_ner_labels}")

# Map integer to string labels for the sentence
example_mapped_labels = [labels_int2str[l] for l in example_ner_labels]
print(f'Labels: {example_mapped_labels}')

# Print the sentence split into tokens.
example_tokenized = tokenizer(example_input_tokens, is_split_into_words=True)
print('BERT Tokenized: ', example_tokenized.tokens())
```

```python
# Print the number of tokens in the vocabulary
print(f'Vocab size: {tokenizer.vocab_size}')

# # Print the sentence mapped to token ids.
print('Token IDs: ', tokenizer.convert_tokens_to_ids(example_tokenized.tokens()))

# Of course, there are now way more tokens than labels! Fortunately the HF tokenizer
# provides a function that will give us the mapping:
print(example_tokenized.word_ids())
```

```python
def labels_tokens_alignment(labels, word_ids):
    new_labels = []  # Initialize a list to store the new labels
    current_word = None  # Variable to keep track of the current word ID
    for word_id in word_ids:  # Iterate over each word ID in the word_ids list
        if word_id != current_word:  # Check if the word ID has changed
            current_word = word_id  # Update the current word ID
            # Append -100 if the word ID is None, otherwise append the corresponding label
            new_labels.append(-100 if word_id is None else labels[word_id])
        else:  # If the word ID is the same as the previous one
            # Append -100 if the word ID is None, otherwise check if the label starts with 'B'
            # If it does, change 'B' to 'I' and append the corresponding label, otherwise append the original label
            new_labels.append(-100 if word_id is None else labels_str2int['I' + labels_int2str[labels[word_id]][1:]]
if labels_int2str[labels[word_id]][0] == 'B' else labels[word_id])
    return new_labels  # Return the list of new labels
```

```python
# Let's check the function on the example from before. The special tokens don't have labels,
# so we'll just replace those with _
aligned_labels = labels_tokens_alignment(example_ner_labels, example_tokenized.word_ids())
print(f"Tokens: {example_tokenized.tokens()}")
print(f"Aligned labels: {[labels_int2str[l] if l >= 0 else '_' for l in aligned_labels]}")
```

```python
# Need to get the whole dataset into this format, so need to write a fn
# we can apply efficiently across all examples using Dataset.map.
def tokenize_and_align_labels(examples):
    # Tokenize the input tokens with truncation and word splitting
    tokenized_inputs = tokenizer(
        examples["tokens"], truncation=True, is_split_into_words=True
    )
    all_labels = examples["ner_tags"]  # Extract the NER tags from the examples
    new_labels = []  # Initialize a list to store the new labels for all examples
    for i, labels in enumerate(all_labels):  # Iterate over each set of labels
        word_ids = tokenized_inputs.word_ids(i)  # Get the word IDs for the current example
        # Align the labels with the tokens and append the result to new_labels
        new_labels.append(labels_tokens_alignment(labels, word_ids))

    tokenized_inputs["labels"] = new_labels  # Add the new labels to the tokenized inputs
    return tokenized_inputs  # Return the tokenized inputs with the new labels
```

```python
# This code trains the model and evaluates it on test data. It should print
# progress messages during training indicating loss, accuracy and training speed.
# You will likely need to make changes to this code for it to work for token classification.
#
# TODO: change this
def train(model,
          train_dataset,
          val_dataset,
          num_epochs,
          batch_size,
          optimizer_cls,
          lr,
          weight_decay,
          device,
          collate_fn=None,
          log_every=100):
  # Set the model to training mode and move it to the specified device
  model = model.train().to(device)
  # Create a DataLoader for the training dataset
  dataloader = DataLoader(train_dataset, batch_size, shuffle=True, collate_fn=collate_fn)

  # Initialize the optimizer based on the specified optimizer class
  if optimizer_cls == 'SGD':
    optimizer = torch.optim.SGD(model.parameters(), lr=lr, weight_decay=weight_decay)
  elif optimizer_cls == 'Adam':
    optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=weight_decay)
  elif optimizer_cls == 'AdamW':
    optimizer = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=weight_decay)

  # Initialize lists to store training and validation metrics
  train_loss_history = []
  train_acc_history = []
  val_loss_history = []
  val_acc_history = []

  # Define the loss function
  lossfn = nn.NLLLoss()
  for e in range(num_epochs):  # Loop over each epoch
    model.train(True)  # Set the model to training mode
    epoch_loss_history = []
    epoch_acc_history = []
    start_time = time.time()
    for i, batch in enumerate(tqdm(dataloader, desc="Training batches")):  # Loop over each batch
      # Move the batch to the specified device
      batch = {k:v.to(device) for k,v in batch.items() if isinstance(v, torch.Tensor)}
      y = batch.pop('labels')  # Extract the labels from the batch
```

```
      logits = model(**batch)  # Forward pass

      # Apply log-softmax to logits before passing to NLLLoss
      log_probs = torch.log_softmax(logits, dim=-1)
      loss = lossfn(log_probs.view(-1, log_probs.size(-1)), y.view(-1))  # Compute the loss

      pred = logits.argmax(dim=-1)  # Get the predictions
      acc = (pred == y).float().mean()  # Compute the accuracy

      epoch_loss_history.append(loss.item())  # Append the loss to the epoch history
      epoch_acc_history.append(acc.item())  # Append the accuracy to the epoch history

      if (i % log_every == 0):  # Log the training progress every 'log_every' iterations
        speed = 0 if i == 0 else log_every/(time.time()-start_time)
        print(f'epoch: {e}\t iter: {i}\t train_loss: {np.mean(epoch_loss_history):.3e}\t
train_acc:{np.mean(epoch_acc_history):.3f}\t speed:{speed:.3f} b/s')
        start_time = time.time()
      loss.backward()  # Backward pass
      optimizer.step()  # Update the model parameters
      optimizer.zero_grad()  # Zero the gradients

    # Evaluate the model on the validation dataset
    val_loss, val_metrics, predictions = run_eval(model, val_dataset, batch_size, device, collate_fn=collate_fn)

    val_acc = val_metrics['overall_accuracy']
    val_p = val_metrics['overall_precision']
    val_r = val_metrics['overall_recall']
    val_f1 = val_metrics['overall_f1']

    # Append the metrics to the history lists
    train_loss_history.append(np.mean(epoch_loss_history))
    train_acc_history.append(np.mean(epoch_acc_history))
    val_loss_history.append(val_loss.item())
    val_acc_history.append(val_acc)
    print(f'epoch: {e}\t train_loss: {train_loss_history[-1]:.3e}\t train_accuracy:{train_acc_history[-1]:.3f}\t
val_loss: {val_loss_history[-1]:.3e}\t val_acc:{val_acc_history[-1]:.3f}\t val_p:{val_p:.3f}\t val_r:{val_r:.3f}\t
val_f1:{val_f1:.3f}')

  # Return the trained model and the training/validation metrics
  return model, (train_loss_history, train_acc_history, val_loss_history, val_acc_history)
```

```
# This code defines the token classification class using BERT.
# The classifier is defined on top of the final layer of BERT.
# The classifier has 1 hidden layer with 128 hidden nodes though we have found that
# using a smaller number of hidden nodes does not make much difference,
#
# TODO: implement this
class BertForTokenClassification(nn.Module):
```

```
  def __init__(self, bert_pretrained_config_name, num_classes, freeze_bert=False, dropout_prob=0.1):
    '''
    BERT with a classification MLP
    args:
    - bert_pretrained_config_name (str): model name from huggingface hub
    - num_classes (int): number of classes in the classification task
    - freeze_bert (bool): [default False] If true gradients are not computed for
                          BERT's parameters.
    - dropout_prob (float): [default 0.1] probability of dropping each activation.
    '''
    super().__init__()
    # Load the pre-trained BERT model from Huggingface hub
    self.bert = BertModel.from_pretrained(bert_pretrained_config_name)
    # Freeze BERT parameters if freeze_bert is True
    self.bert.requires_grad_(not freeze_bert)

    # Define a dropout layer
    self.dropout = nn.Dropout(dropout_prob)
    # Define a classifier with a linear layer
    self.classifier = nn.Sequential(
      nn.Linear(self.bert.config.hidden_size, num_classes)
      # nn.ReLU(),
      # nn.Dropout(dropout_prob),
      # nn.Linear(128, num_classes)
    )

  def forward(self, input_ids, attention_mask=None, token_type_ids=None, labels=None):
    # Pass inputs through BERT model
    outputs = self.bert(input_ids, attention_mask=attention_mask, token_type_ids=token_type_ids)
    # Get the last hidden state from BERT outputs
    sequence_output = outputs.last_hidden_state
    # Apply dropout to the sequence output
    sequence_output = self.dropout(sequence_output)
    # Pass the sequence output through the classifier to get logits
    logits = self.classifier(sequence_output)
    return logits  # Return the logits
```

```
# This is where fine-tuning of the classifier happens.
# Here we are training with batch size 32 for 5 epochs.

# At the end of each epoch, you also see validation loss and validation accuracy.
# Change the device as described above if you will not be using a GPU

# Set the random seed(s) for reproducability
torch.random.manual_seed(8942764)
torch.cuda.manual_seed(8942764)
np.random.seed(8942764)
```

```python
# Make sure this is the same as you use for tokenization!
bert_model = 'bert-base-cased'

num_labels = len(labels_int2str)
print(f"Num labels: {num_labels}")

# conll hyperparams
# multiply your learning rate by k when using batch size of kN
lr = 4*2e-5 # 1e-3
weight_decay = 0.01
epochs = 5
batch_size = 32
dropout_prob = 0.2
freeze_bert = False

bert_cls = BertForTokenClassification(bert_model, num_labels, dropout_prob=dropout_prob, freeze_bert=freeze_bert)

print(f'Trainable parameters: {sum([p.numel() for p in bert_cls.parameters() if p.requires_grad])}\n')

# Flag for setting "debug" mode. Set debug to False for full training.
debug = False

# Sample a subset of the training data for faster iteration in debug mode
subset_size = 1000
subset_indices = torch.randperm(len(tokenized_data_splits['train']))[:subset_size]
train_subset = Subset(tokenized_data_splits['train'], subset_indices)

bert_cls, bert_cls_logs = train(bert_cls, tokenized_data_splits['train'] if not debug else train_subset,
tokenized_data_splits['dev'],
                                num_epochs=epochs, batch_size=batch_size, optimizer_cls='AdamW',
                                lr=lr, weight_decay=weight_decay, device=device,
                                collate_fn=data_collator, log_every=10 if debug else 100)

# Final eval
final_loss, final_metrics, eval_pred = run_eval(bert_cls, tokenized_data_splits['dev'], batch_size=32, device=device,
collate_fn=data_collator)
final_acc = final_metrics['overall_accuracy']
final_p = final_metrics['overall_precision']
final_r = final_metrics['overall_recall']
final_f1 = final_metrics['overall_f1']
print(f'\nFinal Loss: {final_loss:.3e}\t Final Accuracy: {final_acc:.3f}\t dev_p:{final_p:.3f}\t
dev_r:{final_r:.3f}\t dev_f1:{final_f1:.3f}')
```

```python
import json

# Define the output file name for saving the mapped predictions
output_file = "test_predictions_bert.json"
```

```python
# Open the output file in write mode
with open(output_file, "w") as f:
    # Save the test predictions to the JSON file with indentation for readability
    json.dump(test_pred, f, indent=4)

# Print a message indicating that the mapped aligned labels have been saved
print(f"Mapped aligned labels saved to {output_file}")
```

LLM prompting for entity labeling.

```python
# Here is how you can use the API to prompt the OpenAI model.
# Docs: https://platform.openai.com/docs/api-reference
# messages = [
#     {'role': 'system', 'content':
#      """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', 'content': """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', 'content': """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', 'content': """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: """}
# ]

# 1. Few shots...
messages = [
    {'role': 'system', 'content':
     """Label the following text with the given entity types: Deity, Mythological_king, Cretaceous_dinosaur, Aquatic_mammal, Aquatic_animal, Goddess.
     Use tags like '<Cretaceous_dinosaur> Beipiaognathus </Cretaceous_dinosaur>'."""
     },
     {'role': 'user', 'content': """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', 'content': """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', 'content': """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: """}
]

# 2. Chain of thoughts
```

```
messages = [
    {'role': 'system', 'content':
     """Label the following text with the given entity types: Deity, Mythological_king, Cretaceous_dinosaur,
Aquatic_mammal, Aquatic_animal, Goddess.
     Use tags like '<Cretaceous_dinosaur> Beipiaognathus </Cretaceous_dinosaur>'. For each label, explain the
reasoning step by step."""},

    {'role': 'user', 'content': """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', 'content':
     """Reasoning:
     - "Hyperion" is described as a Titan, which is a mythological figure, making it a <Deity>.
     - "Euryphaessa" is described as the mother of Helios (the Sun), aligning with mythological roles, making her a
<Goddess>.
     - "Helios" represents the Sun in mythology, so it is labeled as <Deity>.
     - "Selene" is described as the Moon and associated with mythological figures, making her a <Goddess>.
     - "Eos" is described as the Dawn, also aligning with a mythological role, making her a <Goddess>.

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

    {'role': 'user', 'content':
     """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: """}
]

# 3. Dynamic prompt construction
messages = [
    {'role': 'system', 'content':
     """Label the following text with the given entity types: Deity, Mythological_king, Cretaceous_dinosaur,
Aquatic_mammal, Aquatic_animal, Goddess.
     Use tags like '<Cretaceous_dinosaur> Beipiaognathus </Cretaceous_dinosaur>'. Select examples that closely match
the context or sentence structure of the input text."""},

    # Dynamically selected demonstration 1
    {'role': 'user', 'content': """Text: The well-known dinosaur Tyrannosaurus rex lived during the late Cretaceous
period and was one of the most famous theropods."""},
    {'role': 'system', 'content': """Labels: The well-known dinosaur <Cretaceous_dinosaur> Tyrannosaurus rex
</Cretaceous_dinosaur> lived during the late Cretaceous period and was one of the most famous theropods."""},

    # Dynamically selected demonstration 2
    {'role': 'user', 'content': """Text: Neptune was considered the god of the sea in Roman mythology, often depicted
with a trident."""},
```

```
    {'role': 'system', 'content': """Labels: <Deity> Neptune </Deity> was considered the god of the sea in Roman mythology, often depicted with a trident."""},

    # Dynamically selected demonstration 3
    {'role': 'user', 'content': """Text: Selene was frequently associated with the Moon and was part of Greek mythology's pantheon of deities."""},
    {'role': 'system', 'content': """Labels: <Goddess> Selene </Goddess> was frequently associated with the Moon and was part of Greek mythology's pantheon of deities."""},

    # Input text to label
    {'role': 'user', 'content': """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: """}
]


# # This is where you provide the final prompt that we want the model to complete to give us the answer.
# 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: """

response = client.chat.completions.create(
    model="gpt-3.5-turbo",
    temperature=0.0,
    seed=random_seed,
    messages=messages
)

print(response.choices[0].message.content)

# You can also print out the usage, in number of tokens.
# Pricing is per input/output token, listed here: https://openai.com/pricing
print(f"Usage: {response.usage.prompt_tokens} input, {response.usage.completion_tokens} output, {response.usage.total_tokens} total tokens")
```

```
# Load the dataset
from datasets import Dataset, ClassLabel, Sequence

data_splits = load_dataset('json', data_files={'train': 'dinos_and_deities_train_bio.jsonl', 'dev': 'dinos_and_deities_dev_bio_sm.jsonl', 'test': 'dinos_and_deities_test_bio_nolabels.jsonl'})

# Load dicts for mapping int labels to strings, and vice versa
label_names_fname = "dinos_and_deities_train_bio.jsonl.labels"
labels_int2str = []
with open(label_names_fname) as f:
    labels_int2str = f.read().split()
print(f"Labels: {labels_int2str}")
labels_str2int = {l: i for i, l in enumerate(labels_int2str)}
```

```python
# Also create a set containing the original labels, without B- and I- tags
orig_labels = set()
for label in labels_str2int.keys():
    orig_label = label[2:]
    if orig_label:
        orig_labels.add(orig_label)
print(f"Orig labels: {orig_labels}")

data_splits.cast_column("ner_tags", Sequence(ClassLabel(names=labels_int2str)))
print(data_splits)
```

```python
# Ok, now let's make the prompting a bit more programmatic. First, implement a function that takes an example from
# the dataset, and converts it into a message for the model using the format we specified above.
# You might want to use the Python string "format" function to make this a bit easier, especially since
# You will be experimenting with different prompts later.
#
# TODO: implement this.
def get_message(example):
    """
    Convert a dataset example into the message format expected by the model.

    :param example: Example from the dataset, which should include 'content' and 'ner_strings' fields.
    :return: A string formatted to pass as input to the model, which includes text and its corresponding labels.
    """
    # Extract content (the text to be analyzed) and labels (BIO labels for each token)
    text = example['content']
    tokens = example['tokens']
    ner_labels = example['ner_strings']  # BIO labels for each token

    # Return the formatted message to pass to the model
    message = f"Text: {text}"
    return message
```

```python
# Next we're going to implement a function to return the chat_history, but in order to do that we first need
# to be able to convert labeled examples from the dataset into a format that makes more sense for the model,
# in this case the HTML-style format we specified in the example. That's the task for this function: take
# an example from the dataset as input, and return a string that has tagged the text with labels in the given
# HTML-style format.
#
# TODO: implement this.
def convert_bio_to_prompt(example):
    """
    Convert the BIO-labeled text into a tagged format for model input.

    :param example: A dataset example with 'tokens' and 'ner_tags' fields.
    :return: A string with the text, tagged with labels in the specified format.
```

```
    """
    tokens = example['tokens']  # List of tokens from the example
    ner_tags = example['ner_tags']  # List of BIO tags corresponding to each token

    # Start with the text being empty
    text = ""

    for token, tag_id in zip(tokens, ner_tags):
        # Get the string label from the integer tag id
        tag = labels_int2str[tag_id]

        # If the tag starts with 'B-' or 'I-', it's an entity
        if tag.startswith('B-'):
            entity_type = tag[2:]  # Get the entity type (e.g., 'Deity', 'Cretaceous_dinosaur')
            text += f" <{entity_type}>{token}</{entity_type}>"
        elif tag.startswith('I-'):
            entity_type = tag[2:]  # Continuation of the entity
            text += f" <{entity_type}>{token}</{entity_type}>"
        else:
            text += f" {token}"

    return text
```

```
# 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):
    """
    Generate a list of prompt examples for the model, using `shots` as the number of demonstration examples.

    :param shots: Number of examples to include for few-shot learning.
    :param dataset: The dataset to sample examples from.
    :param entity_types_list: List of entity types (e.g., ['Deity', 'Cretaceous_dinosaur']).
    :param convert_bio_to_prompt_fn: Function to convert BIO examples to a string.
    :return: List of message dictionaries for the model's chat history.
    """
    chat_history = []

    # First, add the system message that introduces the entity types
    chat_history.append({
        'role': 'system',
        'content': 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: Deity, Mythological_king, Cretaceous_dinosaur,
Aquatic_mammal, Aquatic_animal, Goddess.\
```

```
                     Label the enities by surrounding them with tags like '<Cretaceous_dinosaur> Beipiaognathus
</Cretaceous_dinosaur>'."
    })

    # Now, add the user messages based on the dataset examples
    for i in range(shots):
        example = dataset[i]
        print("Example:", example)
        formatted_example = convert_bio_to_prompt_fn(example)
        print("Example:", formatted_example)

        # Example 1: Show the text with expected labels
        chat_history.append({
            'role': 'user',
            'content': f"Text: {example['content']}\nLabels: {formatted_example}"
        })

    return chat_history
```

```
# Chain-of-Thought Implementation

def get_chat_history(shots, dataset, entity_types_list, convert_bio_to_prompt_fn):
    """
    Generate a list of prompt examples for the model, using `shots` as the number of demonstration examples.

    :param shots: Number of examples to include for few-shot learning.
    :param dataset: The dataset to sample examples from.
    :param entity_types_list: List of entity types (e.g., ['Deity', 'Cretaceous_dinosaur']).
    :param convert_bio_to_prompt_fn: Function to convert BIO examples to a string.
    :return: List of message dictionaries for the model's chat history.
    """
    chat_history = []

    # First, add the system message that introduces the entity types and explains the chain of thought process
    chat_history.append({
        'role': 'system',
        'content': 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: {', '.join(entity_types_list)}.\
                     For each example, think step-by-step and provide detailed reasoning before labeling the
entities.\
                     Label the entities by surrounding them with tags like '<Cretaceous_dinosaur> Beipiaognathus
</Cretaceous_dinosaur>'."
    })

    # Now, add the user messages based on the dataset examples
    for i in range(shots):
        example = dataset[i]
        print("Example:", example)
```

```
        formatted_example = convert_bio_to_prompt_fn(example)
        print("Example:", formatted_example)

        # Example 1: Show the text with expected labels and include reasoning steps
        chat_history.append({
            'role': 'user',
            'content': f"Text: {example['content']}\n"
                       f"Step 1: Identify the entities in the text.\n"
                       f"Step 2: Determine the type of each entity based on the context.\n"
                       f"Step 3: Label the entities by surrounding them with the appropriate tags.\n"
                       f"Labels: {formatted_example}"
        })

    return chat_history
```

```
import random

def get_chat_history(shots, dataset, entity_types_list, convert_bio_to_prompt_fn, input_text):
    """
    Generate a dynamic prompt with contextually relevant examples for the model.

    :param shots: Number of examples to include for few-shot learning.
    :param dataset: The dataset to sample examples from.
    :param entity_types_list: List of entity types (e.g., ['Deity', 'Cretaceous_dinosaur']).
    :param convert_bio_to_prompt_fn: Function to convert BIO examples to a string.
    :param input_text: The input text for which the prompt is being constructed.
    :return: List of message dictionaries for the model's chat history.
    """
    chat_history = []

    # Add the system message introducing the task
    chat_history.append({
        'role': 'system',
        'content': f"""Label the following text with the given entity types: {', '.join(entity_types_list)}.
                    Use tags like '<Cretaceous_dinosaur> Beipiaognathus </Cretaceous_dinosaur>'.
                    Select examples that closely match the context or sentence structure of the input text."""
    })

    # Select contextually relevant examples dynamically
    selected_examples = []
    for example in dataset:
        if len(selected_examples) >= shots:
            break
        # Simple heuristic: check if input text shares entity types or structure with the example
        if any(entity in example['content'] for entity in input_text.split()):
            selected_examples.append(example)

    # If not enough examples are found, pad with random examples
```

```
    while len(selected_examples) < shots:
        selected_examples.append(random.choice(dataset))

    # Add the dynamically selected examples to the chat history
    for example in selected_examples:
        formatted_example = convert_bio_to_prompt_fn(example)
        chat_history.append({
            'role': 'user',
            'content': f"Text: {example['content']}"
        })
        chat_history.append({
            'role': 'system',
            'content': f"Labels: {formatted_example}"
        })

    # Add the input text for labeling
    chat_history.append({
        'role': 'user',
        'content': f"Text: {input_text}\nLabels: "
    })

    return chat_history
```

```
# Now let's wrap that call in a function that takes shots and an example, calls the API and returns the response.
def call_api_openai(shots, example):
    success = False
    while not success:
        try:
            chat_history = get_chat_history(shots, data_splits['train'], orig_labels, convert_bio_to_prompt)
            message = {'role': USER_STR, 'content': get_message(example)}
            chat_history.append(message)
            response = client.chat.completions.create(
                model="gpt-3.5-turbo",
                temperature=0.0,
                messages=chat_history
            )
            success = 1
        except Exception as err:
            tqdm.write(f"Caught exception: {err}")
    return response.choices[0].message.content
```

```
# Now we want to be able to evaluate the model, in order to compare it to e.g. the fine-tuned BERT model.
# In order to do this, we need to write the reverse of the convert_bio_to_prompt function, so that we can
# convert in the other direction, from the generated response in prompt format, back to bio for evaluation
# using seqeval.
#
# The input to this function is the string response from the model, and the output should be a list of
```

```
# text BIO labels corresponding to the labeling implied by the tagged output produced by the model, as
# well as the list of tokens (since the generative model could return something different than we gave it,
# and we need to handle that somehow in the eval).
#
# TODO: implement this
import re

def convert_response_to_bio(response):
    """
    Convert a model's generated response with HTML-style tags into BIO format.

    :param response: The string response from the model with tagged entities in HTML format.
    :return: A tuple containing two lists:
        - bio_labels: The list of BIO labels corresponding to each token.
        - tokens: The list of tokens corresponding to each entity or non-entity.
    """
    start_labels = r"^Labels:"
    response =re.sub(start_labels, "", response).strip()

    tag_pattern = r"<(/?)([a-zA-Z_]+)>([^<]*)"
    punctuation_pattern = rf"^[{re.escape(string.punctuation)}]+$"


    # Initialize variables
    labels = []
    tokens = []

    # Split the response into tokens and tags
    for match_idx, match in enumerate(re.finditer(tag_pattern, response)):
        if match_idx == 0 and match.start() != 0:
            text = response[:match.start()].strip()
            texts = text.split(" ")
            for t in texts:
                tokens.append(t)
                labels.append("O")

        # Extract the tag and text
        tag, entity, text = match.groups()
        text = text.strip()

        # Split the text into tokens
        text_tokens = text.split(" ")

        text_tokens_no_punctuation = []

        for i, token in enumerate(text_tokens):
            if re.match(punctuation_pattern, token):
                if i == 0:
```

```python
                tokens[-1] = tokens[-1] + token
            else:
                if len(text_tokens_no_punctuation) == 0:
                    labels[-1] = labels[-1] + token
                else:
                    text_tokens_no_punctuation[-1] = text_tokens_no_punctuation[-1] + token
        else:
            text_tokens_no_punctuation.append(token)

    # Add the tokens and labels
    for i, token in enumerate(text_tokens_no_punctuation):
        if token:
            tokens.append(token)
            if tag == "/":
                labels.append("O")
            elif i == 0:
                labels.append(f"B-{entity}")
            else:
                labels.append(f"I-{entity}")

  return labels, tokens
```

```python
# Now we can put all of the above together to evaluate!
metric = evaluate.load("seqeval")

def run_eval_test(dataset, shots):
  pred = []
  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)

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

      # 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)
    pred.append(predictions)



return pred
```