Understanding and Implementing Medprompt

Digging into the details behind the prompting framework

Anand Subramanian · Follow

Published in Towards Data Science · 14 min read · Jul 6, 2024

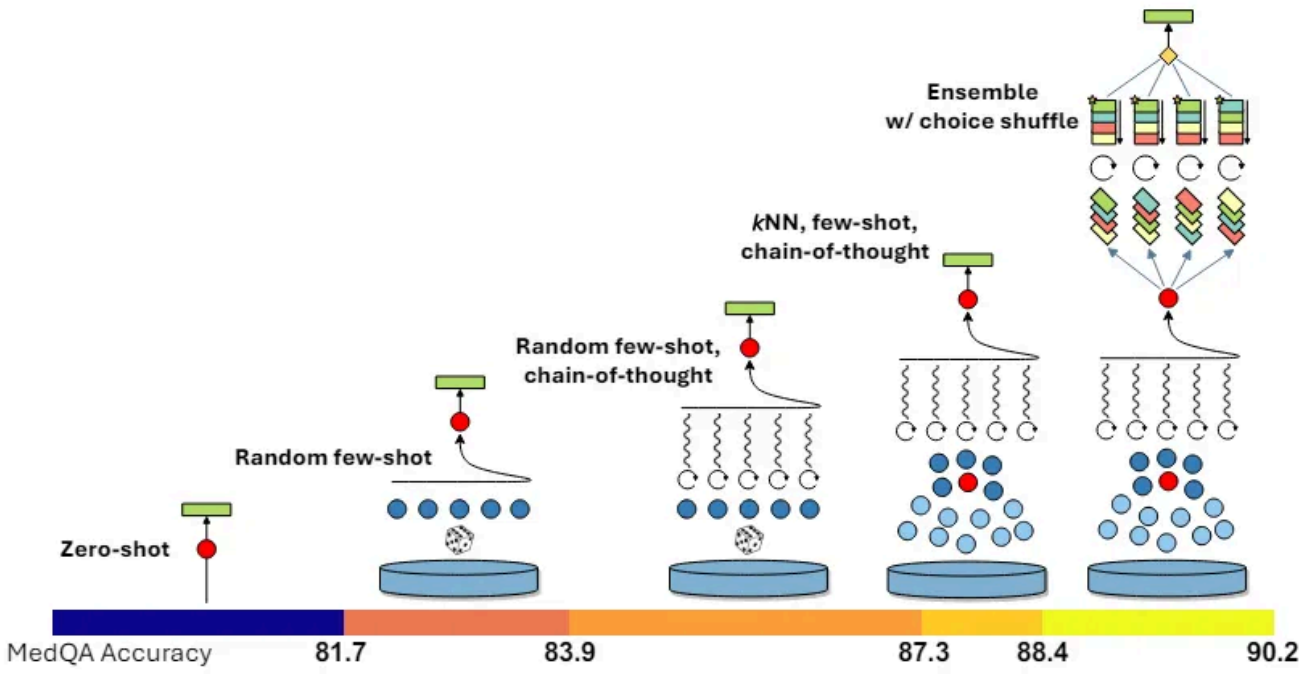


Illustration of the various components of the Medprompt Strategy (Image taken from Fig:6 from the Medprompt paper [1] (https://arxiv.org/abs/2311.16452)

In my first blog post, I explored prompting and its significance in the context of Large Language Models (LLMs). Prompting is crucial for obtaining high-quality outputs from LLMs, as it guides the model's responses and ensures they are relevant to the task at hand. Building on that foundation, two crucial questions often arise when trying to solve a use case using LLMs: how far can you push performance with prompting alone, and when do you bite the bullet and decide it might be more effective to fine-tune a model instead?

When making design decisions about leveraging prompting, several considerations come into play. Techniques like few-shot prompting and Chain-of-Thought (CoT) [2] prompting can help in boosting the performance of LLMs for most tasks. Retrieval-Augmented Generation (RAG) pipelines can further enhances LLM performance by adapting to new domains without fine-tuning and providing controllability over grounding the generated outputs while reducing hallucinations. Overall, we have a suite of tools to push the needle in terms of LLM performance without explicitly resorting to fine-tuning.

Fine-tuning comes with its own set of challenges and complications, in terms of labelled data requirements and the costs associated with training of LLMs and their deployment. Fine-tuning may also increase the hallucinations of the LLM in certain cases [3]. Putting this all together, we can see that there is significant value in trying to optimize LLM performance for our task through prompting before resorting to fine-tuning.

So, how do we go about this? In this article, we explore Medprompt [1], a sophisticated prompting strategy introduced by Microsoft. Medprompt ties together principles from few-shot prompting, CoT prompting and RAG to

enhance the performance of GPT-4 in the healthcare domain without any domain-specific fine-tuning.

Table of Contents:

1. MedPrompt Explained
2. Components of MedPrompt
3. Implementing MedPrompt
4. Evaluating Performance
5. Conclusion
6. References

MedPrompt Explained

LLMs have demonstrated impressive capabilities across various sectors, particularly in healthcare. Last year, Google introduced MedPaLM [4] and MedPaLM-2 [5], LLMs that not only excel in Medical Multiple-Choice Question Answering (MCQA) datasets but also perform competitively and even outperform clinicians in open-ended medical question answering . These models have been tailored specifically for the healthcare domain through instruction fine-tuning and the use of clinician-written Chain-of-Thought templates, significantly enhancing their performance.

In this context, the paper **"Can Generalist Foundation Models Outcompete Special-Purpose Tuning? Case Study in Medicine"** [1] from Microsoft raises a compelling question:

> Can the performance of a generalist model like GPT-4 be improved for a specific domain without relying on

domain-specific fine-tuning or expert-crafted resources?

As part of this study, the paper introduces **Medprompt**, an innovative prompting strategy that not only improves the model's performance but also surpasses specialized models such as MedPaLM-2.

Table 1: Performance of different foundation models on multiple choice components of MultiMedQA [29]. GPT-4 with Medprompt outperforms all other models on every benchmark.

Dataset	Flan-PaLM 540B* (choose best)	Med-PaLM 2* (choose best)	GPT-4 (5 shot)	GPT-4 (Medprompt)
MedQA				
US (4-option)	67.6	86.5	81.4	**90.2****
PubMedQA				
Reasoning Required	79.0	81.8	75.2	**82.0**
MedMCQA				
Dev	57.6	72.3	72.4	**79.1**
MMLU				
Clinical Knowledge	80.4	88.7	86.4	**95.8**
Medical Genetics	75.0	92.0	92.0	**98.0**
Anatomy	63.7	84.4	80.0	**89.6**
Professional Medicine	83.8	**95.2**	93.8	**95.2**
College Biology	88.9	95.8	95.1	**97.9**
College Medicine	76.3	83.2	76.9	**89.0**

* Sourced directly from [29] and [30]. "Choose best" refers to a process used in the Med-Palm studies of executing several distinct approaches and selecting the best performing strategy for each dataset among the variety of experimental methods tried. Flan-PaLM 540B and Med-PaLM 2 are also both fine-tuned on subsets of these benchmark datasets. By contrast, every GPT-4 reported number uses a single, consistent strategy across all datasets.

** We achieve 90.6%, as discussed in Section 5.2, with $k = 20$ and 11x ensemble steps. The 90.2% represents "standard" Medprompt performance with $k = 5$ few shot examples and a 5x ensemble.

Comparison of various LLMs on medical knowledge benchmarks. GPT-4 with Medprompt outperforms Med-PaLM 2 across all these datasets. (Image of Table 1 from the Medprompt paper [1] (https://arxiv.org/abs/2311.16452))

GPT-4 with Medprompt outperforms Med-PaLM 2 across all medical MCQA benchmarks without any **domain-specific fine-tuning.** Let's explore the components in Medprompt.

Components of Medprompt

Medprompt ties together principles from few-shot prompting, CoT prompting and RAG. Specifically there are 3 components in this pipeline:

Dynamic Few-shot Selection `

Few-shot prompting refers to utilizing example input-output pairs as context for prompting the LLM. If these few-shot samples are static, the downside is that they may not be the most relevant examples for the new input. **Dynamic Few-shot Selection,** the first component in Medprompt, helps overcome this by selecting the few-shot examples based on each new task input. This method involves training a K-Nearest Neighbors (K-NN) algorithm on the training set, which then retrieves the most similar training set examples to the test input based on cosine similarity in an embedding space. This strategy efficiently utilizes the existing training dataset to retrieve relevant few-shot examples for prompting the LLM.

Self-Generated CoT

As noted in the paper [1], CoT traditionally relies on manually crafted few-shot exemplars that include detailed reasoning steps, as used with MedPaLM-2, where such prompts were written by medical professionals. Medprompt introduces **Self-Generated CoT** as the second module, where the LLM is used to produce detailed, step-by-step explanations of its reasoning process, culminating in a final answer choice. By automatically generating CoT reasoning steps for each training datapoint, the need for manually crafted exemplars is bypassed. To ensure that only correct predictions with

reasoning steps are retained and incorrect responses are filtered out, the answer generated by GPT-4 is cross-verified against the ground truth.

Choice Shuffling Ensemble

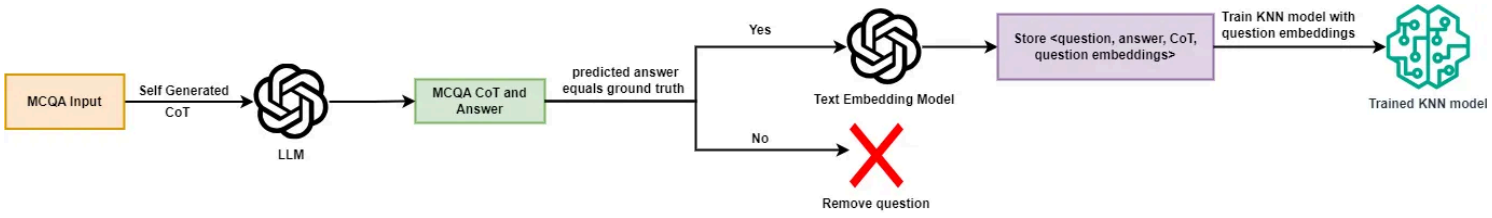
The Choice Shuffling Ensemble technique is the third technique introduced by Medprompt. It is designed to combat the inherent biases that may affect the model's decision-making process, particularly position bias in multiple-choice settings. The ordering of the answer choices is shuffled, and this process is repeated k times to create k variants of the same question with shuffled answer choices. During inference, each variant is used to generate an answer, and a majority vote is performed over all variants to pick the final predicted option.

How are these components used in the preprocessing and inference stage?

Let's now have a look at the preprocessing and inference stages in Medprompt.

Preprocessing Stage

In the preprocessing pipeline, we begin by taking each question from the training dataset and incorporating detailed instructions within the prompt to guide the generation of both an answer and its associated reasoning steps. The LLM is prompted to generate the answer and reasoning steps. After obtaining the generated response, we verify its accuracy by comparing the predicted answer to the ground truth for that particular question.

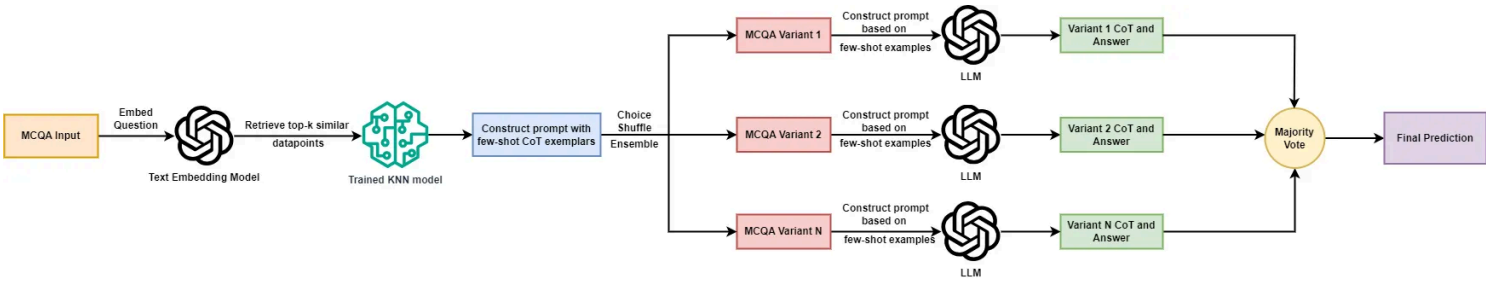


Medprompt Preprocessing Pipeline (Image by Author)

If the prediction is incorrect, we exclude this instance from our database of relevant questions. If the prediction is correct, we proceed by embedding the question using a text embedding model. We then store the question, question embedding, answer, and Chain of Thought (CoT) reasoning in a buffer. Once all questions have been processed, we utilize the embeddings for training a KNN model. This trained KNN model acts as our retriever in a RAG pipeline, enabling us to efficiently query and retrieve the top-k similar data points based on cosine similarity within the embedding space.

Inference Pipeline

During the inference stage, each question from our test set is first embedded using the text embedding model. We then utilize the KNN model to identify the top-k most similar questions. For each retrieved data point, we have access to the self-generated Chain of Thought (CoT) reasoning and the predicted answer. We format these elements — question, CoT reasoning, and answer — into few-shot examples for our eventual prompt.



Medprompt Inference Pipline (Image by Author)

We now perform **choice shuffling ensembling** by shuffling the order of answer choices for each test question, creating multiple variants of the same question. The LLM is then prompted with these variants, along with the

corresponding few-shot exemplars, to generate reasoning steps and an answer for each variant. Finally, we perform a majority vote over the predictions from all variants and select the final prediction.

Implementing Medprompt

> *The code related to this implementation can be found at this github repo link.*

We use the MedQA [6] dataset for implementing and evaluating Medprompt. We first define helper functions for parsing the jsonl files.

```
def write_jsonl_file(file_path, dict_list):
    """
    Write a list of dictionaries to a JSON Lines file.

    Args:
    - file_path (str): The path to the file where the data will be written.
    - dict_list (list): A list of dictionaries to write to the file.
    """
    with open(file_path, 'w') as file:
        for dictionary in dict_list:
            json_line = json.dumps(dictionary)
            file.write(json_line + '\n')

def read_jsonl_file(file_path):
    """
    Parses a JSONL (JSON Lines) file and returns a list of dictionaries.

    Args:
        file_path (str): The path to the JSONL file to be read.

    Returns:
        list of dict: A list where each element is a dictionary representing
            a JSON object from the file.
    """
    jsonl_lines = []
    with open(file_path, 'r', encoding="utf-8") as file:
        for line in file:
            json_object = json.loads(line)
            jsonl_lines.append(json_object)
```

```
    return jsonl_lines
```

Implementing Self-Generated CoT

For our implementation, we utilize the training set from MedQA. We implement a zero-shot CoT prompt and process all the training questions. We use **GPT-4o** in our implementation. For each question, we generate the CoT and the corresponding answer. We define a prompt which is based on the template provided in the Medprompt paper.

```
system_prompt = """You are an expert medical professional. You are provided with
Your goal is to think through the question carefully and explain your reasoning
Respond only with the reasoning steps and answer as specified below.
Below is the format for each question and answer:

Input:
## Question: {{question}}
{{answer_choices}}

Output:
## Answer
(model generated chain of thought explanation)
Therefore, the answer is [final model answer (e.g. A,B,C,D)]"""
```

```
def build_zero_shot_prompt(system_prompt, question):
    """
    Builds the zero-shot prompt.

    Args:
        system_prompt (str): Task Instruction for the LLM
        content (dict): The content for which to create a query, formatted as
            required by `create_query`.

    Returns:
        list of dict: A list of messages, including a system message defining
            the task and a user message with the input question.
    """
```

```
    messages = [{"role": "system", "content": system_prompt},
                {"role": "user", "content": create_query(question)}]
    return messages


def build_few_shot_prompt(system_prompt, question, examples, include_cot=True):
    """
    Builds the few-shot prompt.

    Args:
        system_prompt (str): Task Instruction for the LLM
        content (dict): The content for which to create a query, formatted as
            required by `create_query`.

    Returns:
        list of dict: A list of messages, including a system message defining
            the task and a user message with the input question.
    """
    messages = [{"role": "system", "content": system_prompt}]

    for elem in examples:
        messages.append({"role": "user", "content": create_query(elem)})
        if include_cot:
            messages.append({"role": "assistant", "content": format_answer(elem[
        else:
            answer_string = f"""## Answer\nTherefore, the answer is {elem["answe
            messages.append({"role": "assistant", "content": answer_string})

    messages.append({"role": "user", "content": create_query(question)})
    return messages

def get_response(messages, model_name, temperature = 0.0, max_tokens = 10):
    """
    Obtains the responses/answers of the model through the chat-completions API.

    Args:
        messages (list of dict): The built messages provided to the API.
        model_name (str): Name of the model to access through the API
        temperature (float): A value between 0 and 1 that controls the randomnes
        A temperature value of 0 ideally makes the model pick the most likely to
        max_tokens (int): Maximum number of tokens that the model should generat

    Returns:
        str: The response message content from the model.
    """
    response = client.chat.completions.create(
        model=model_name,
        messages=messages,
        temperature=temperature,
        max_tokens=max_tokens
```

```
    )
    return response.choices[0].message.content
```

We also define helper functions for parsing the reasoning and the final answer option from the LLM response.

```
def matches_ans_option(s):
    """
    Checks if the string starts with the specific pattern 'Therefore, the answer

    Args:
    s (str): The string to be checked.

    Returns:
    bool: True if the string matches the pattern, False otherwise.
    """
    return bool(re.match(r'^Therefore, the answer is [A-Z]', s))

def extract_ans_option(s):
    """
    Extracts the answer option (a single capital letter) from the start of the s

    Args:
    s (str): The string containing the answer pattern.

    Returns:
    str or None: The captured answer option if the pattern is found, otherwise N
    """
    match = re.search(r'^Therefore, the answer is ([A-Z])', s)
    if match:
        return match.group(1)  # Returns the captured alphabet
    return None

def matches_answer_start(s):
    """
    Checks if the string starts with the markdown header '## Answer'.

    Args:
    s (str): The string to be checked.

    Returns:
    bool: True if the string starts with '## Answer', False otherwise.
    """
```

```
    return s.startswith("## Answer")

def validate_response(s):
    """
    Validates a multi-line string response that it starts with '## Answer' and e

    Args:
    s (str): The multi-line string response to be validated.

    Returns:
    bool: True if the response is valid, False otherwise.
    """
    file_content = s.split("\n")

    return matches_ans_option(file_content[-1]) and matches_answer_start(s)

def parse_answer(response):
    """
    Parses a response that starts with '## Answer', extracting the reasoning and

    Args:
    response (str): The multi-line string response containing the answer and rea

    Returns:
    tuple: A tuple containing the extracted CoT reasoning and the answer choice.
    """
    split_response = response.split("\n")
    assert split_response[0] == "## Answer"
    cot_reasoning = "\n".join(split_response[1:-1]).strip()
    ans_choice = extract_ans_option(split_response[-1])
    return cot_reasoning, ans_choice
```

We now process the questions in the training set of MedQA. We obtain CoT responses and answers for all questions and store them to a folder.

```
train_data = read_jsonl_file("data/phrases_no_exclude_train.jsonl")

cot_responses = []
os.mkdir("cot_responses")

for idx, item in enumerate(tqdm(train_data)):
    prompt = build_zero_shot_prompt(system_prompt, item)
    try:
```

```
            response = get_response(prompt, model_name="gpt-4o", max_tokens=500)
            cot_responses.append(response)
            with open(os.path.join("cot_responses", str(idx) + ".txt"), "w", encodin
                f.write(response)
        except Exception as e :
            print(str(e))
            cot_responses.append("")
```

We now iterate across all the generated responses to check if they are valid and adhere to the prediction format defined in the prompt. We discard responses that do not conform to the required format. After that, we check the predicted answers against the ground truth for each question and only retain questions for which the predicted answers match the ground truth.

```
questions_dict = []
ctr = 0
for idx, question in enumerate(tqdm(train_data)):
    file = open(os.path.join("cot_responses/", str(idx) + ".txt"), encoding="utf
    if not validate_response(file):
        continue

    cot, pred_ans = parse_answer(file)

    dict_elem = {}
    dict_elem["idx"] = idx
    dict_elem["question"] = question["question"]
    dict_elem["answer"] = question["answer"]
    dict_elem["options"] = question["options"]
    dict_elem["cot"] = cot
    dict_elem["pred_ans"] = pred_ans
    questions_dict.append(dict_elem)

filtered_questions_dict = []
for item in tqdm(questions_dict):
    pred_ans = item["options"][item["pred_ans"]]
    if pred_ans == item["answer"]:
        filtered_questions_dict.append(item)
```

Implementing the KNN model

Having processed the training set and obtained the CoT response for all these questions, we now embed all questions using the **text-embedding-ada-002** from OpenAI.

```
def get_embedding(text, model="text-embedding-ada-002"):
    return client.embeddings.create(input = [text], model=model).data[0].embeddi

for item in tqdm(filtered_questions_dict):
    item["embedding"] = get_embedding(item["question"])
    inv_options_map = {v:k for k,v in item["options"].items()}
    item["answer_idx"] = inv_options_map[item["answer"]]
```


the training set that are most similar to the question from the test set.

```
import numpy as np
from sklearn.neighbors import NearestNeighbors

embeddings = np.array([d["embedding"] for d in filtered_questions_dict])
indices = list(range(len(filtered_questions_dict)))

knn = NearestNeighbors(n_neighbors=5, algorithm='auto', metric='cosine').fit(emb
```

Implementing the Dynamic Few-Shot and Choice Shuffling Ensemble Logic

We can now run inference. We subsample 500 questions from the MedQA test set for our evaluation. For each question, we retrieve the 5 most similar questions from the train set using the KNN module, along with their

respective CoT reasoning steps and predicted answers. We construct a few-shot prompt using these examples.

For each question, we also shuffle the order of the options 5 times to create different variants. We then utilize the constructed few-shot prompt to get the predicted answer for each of the variants with shuffled options.

```
def shuffle_option_labels(answer_options):
    """
    Shuffles the options of the question.

    Parameters:
    answer_options (dict): A dictionary with the options.

    Returns:
    dict: A new dictionary with the shuffled options.
    """
    options = list(answer_options.values())
    random.shuffle(options)
    labels = [chr(i) for i in range(ord('A'), ord('A') + len(options))]
    shuffled_options_dict = {label: option for label, option in zip(labels, opti

    return shuffled_options_dict
```

```
test_samples = read_jsonl_file("final_processed_test_set_responses_medprompt.jso

for question in tqdm(test_samples, colour ="green"):
    question_variants = []
    prompt_variants = []
    cot_responses = []
    question_embedding = get_embedding(question["question"])
    distances, top_k_indices = knn.kneighbors([question_embedding], n_neighbors=
    top_k_dicts = [filtered_questions_dict[i] for i in top_k_indices[0]]
    question["outputs"] = []

    for idx in range(5):
        question_copy = question.copy()
        shuffled_options = shuffle_option_labels(question["options"])
        inv_map = {v:k for k,v in shuffled_options.items()}
```

```
            question_copy["options"] = shuffled_options
            question_copy["answer_idx"] = inv_map[question_copy["answer"]]
            question_variants.append(question_copy)
            prompt = build_few_shot_prompt(system_prompt,  question_copy, top_k_dict
            prompt_variants.append(prompt)

        for prompt in tqdm(prompt_variants):
            response = get_response(prompt, model_name="gpt-4o", max_tokens=500)
            cot_responses.append(response)

        for question_sample, answer in zip(question_variants, cot_responses):
            if validate_response(answer):
                cot, pred_ans = parse_answer(answer)

            else:
                cot = ""
                pred_ans = ""

            question["outputs"].append({"question": question_sample["question"], "op
```

We now evaluate the results of Medprompt over the test set. For each question, we have five predictions generated through the ensemble logic. We take the mode, or most frequently occurring prediction, for each question as the final prediction and evaluate the performance. Two edge cases are possible here:

1. Two different answer options are predicted two times each, with no clear winner.
2. There is an error with the response generated, meaning that we don't have a predicted answer option.

For both of these edge cases, we consider the question to be wrongly answered by the LLM.

```
def find_mode_string_list(string_list):
    """
    Finds the most frequently occurring strings.

    Parameters:
    string_list (list of str): A list of strings.
    Returns:
    list of str or None: A list containing the most frequent string(s) from the
                         Returns None if the input list is empty.
    """
    if not string_list:
        return None

    string_counts = Counter(string_list)
    max_freq = max(string_counts.values())
    mode_strings = [string for string, count in string_counts.items() if count =
    return mode_strings

ctr = 0
for item in test_samples:
    pred_ans = [x["pred_ans"] for x in item["outputs"]]
    freq_ans = find_mode_string_list(pred_ans)

    if len(freq_ans) > 1:
        final_prediction = ""
    else:
        final_prediction = freq_ans[0]

    if final_prediction == item["answer"]:
        ctr +=1

print(ctr / len(test_samples))
```

Evaluating Performance

We evaluate the performance of Medprompt with GPT-4o in terms of accuracy on the MedQA test subset. Additionally, we benchmark the performance of Zero-shot prompting, Random Few-Shot prompting, and Random Few-Shot with CoT prompting.

Method	Accuracy
Zero-Shot	73.2
Random Few-Shot	85.8
Random Few-Shot CoT	90.0
Medprompt	89.4

Results of our evaluation (Image by Author)

We observe that Medprompt and Random Few-Shot CoT prompting outperform the Zero and Few-Shot prompting baselines. However, surprisingly, we notice that Random Few-Shot CoT outperforms our Medprompt performance. This could be due to a couple of reasons:

1. The original Medprompt paper benchmarked the performance of GPT-4. We observe that GPT-4o outperforms GPT-4T and GPT-4 on various text benchmarks significantly (https://openai.com/index/hello-gpt-4o/), indicating that Medprompt could have a lesser effect on a stronger model like GPT-4o.
2. We restrict our evaluation to 500 questions subsampled from MedQA. The Medprompt paper evaluates other Medical MCQA datasets and the full version of MedQA. Evaluating GPT-4o on the complete versions of the datasets could give a better picture of the overall performance.

Conclusion

Medprompt is an interesting framework for creating sophisticated prompting pipelines, particularly for adapting a generalist LLM to a specific domain without the need for fine-tuning. It also highlights the considerations involved in deciding between prompting and fine-tuning for

various use cases. Exploring how far prompting can be pushed to enhance LLM performance is important, as it offers a resource and cost-efficient alternative to fine-tuning.

References:

[1] Nori, H., Lee, Y. T., Zhang, S., Carignan, D., Edgar, R., Fusi, N., ... & Horvitz, E. (2023). Can generalist foundation models outcompete special-purpose tuning? case study in medicine. *arXiv preprint arXiv:2311.16452*. (https://arxiv.org/abs/2311.16452)

[2] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., ... & Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, *35*, 24824–24837. (https://openreview.net/pdf?id=_VjQlMeSB_J)

[3] Gekhman, Z., Yona, G., Aharoni, R., Eyal, M., Feder, A., Reichart, R., & Herzig, J. (2024). Does Fine-Tuning LLMs on New Knowledge Encourage Hallucinations?. *arXiv preprint arXiv:2405.05904*. (https://arxiv.org/abs/2405.05904)

[4] Singhal, K., Azizi, S., Tu, T., Mahdavi, S. S., Wei, J., Chung, H. W., ... & Natarajan, V. (2023). Large language models encode clinical knowledge. *Nature*, *620*(7972), 172–180. (https://www.nature.com/articles/s41586-023-06291-2)

[5] Singhal, K., Tu, T., Gottweis, J., Sayres, R., Wulczyn, E., Hou, L., ... & Natarajan, V. (2023). Towards expert-level medical question answering with large language models. *arXiv preprint arXiv:2305.09617*. (https://arxiv.org/abs/2305.09617)

[6] Jin, D., Pan, E., Oufattole, N., Weng, W. H., Fang, H., & Szolovits, P. (2021). What disease does this patient have? a large-scale open domain question answering dataset from medical exams. *Applied Sciences*, *11*(14), 6421. (https://arxiv.org/abs/2009.13081) (Original dataset is released under a MIT License)

ChatGPT | Large Language Models | Deep Learning | Healthcare | Editors Pick

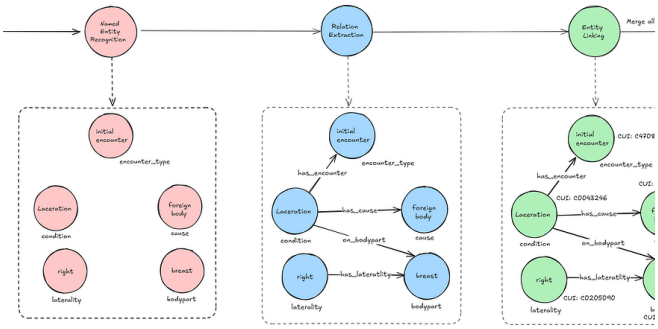
R

Rohit
Jul 8, 2024

what if the self - generated predicts correct answer but its reasoning is false? i would love to hear thoughts on this :)

6 1 reply Reply

More from Anand Subramanian and Towards Data Science

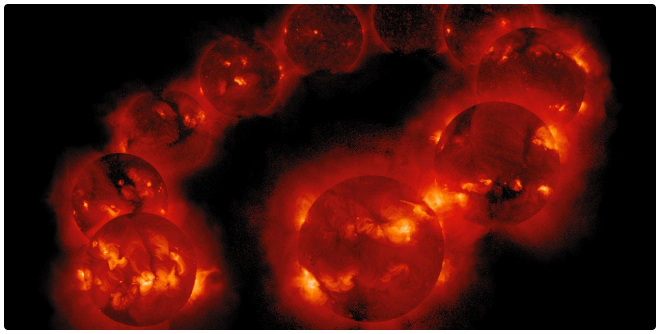


In AI Advances by Anand Subramanian

Creating a Knowledge Graph for ICD Codes using LLMs

Representing ICD Codes as a Knowledge Graph

Nov 5, 2024 399 7



In Towards Data Science by Pau Blasco i Roca

The Solar Cycle(s): history, data analysis and trend forecasting.

A brief article on the Solar Cycles: data analysis and time series forecasting for the...

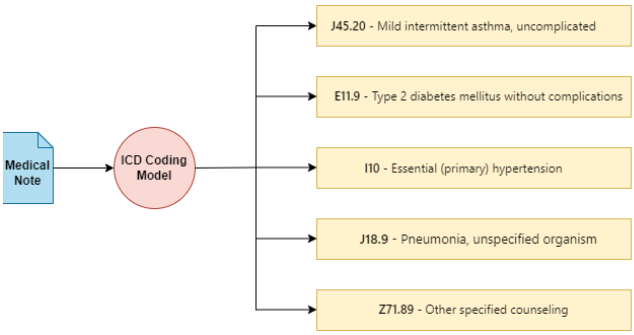
4d ago 164 3

tds In Towards Data Science by Javier Marin

Understanding Emergent Capabilities in LLMs: Lessons fro...

How natural systems fundamental laws help explain AI's unexpected abilities

5d ago 172 6



tds In Towards Data Science by Anand Subramanian

Exploring LLMs for ICD Coding — Part 1

Building automated clinical coding systems with LLMs

May 17, 2024 481 7

See all from Anand Subramanian

See all from Towards Data Science

Recommended from Medium

In Generative AI by Jim Clyde Monge

How To Install And Use DeepSeek R-1 In Your Local PC

Here's a step-by-step guide on how you can run DeepSeek R-1 on your local machine eve...

4d ago 692 12

Alberto Romero

DeepSeek Is Chinese But Its AI Models Are From Another Planet

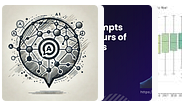
OpenAI and the US are in deep trouble

5d ago 2.3K 51

Lists

The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 545 saves



ChatGPT prompts

51 stories · 2508 saves

What is ChatGPT?

9 stories · 497 saves

ChatGPT

21 stories · 954 saves

In Coding Nexus by DataScience Nexus

Docling : Transform any document into LLM ready data in just a few...

In today's fast-paced world, data is the backbone of innovation. From academic...

Jan 9 31

In GoPenAI by Paras Madan

Building a Multi-Agent System for writing a Book: Crew AI - Tutorial ...

AI agents are becoming the next big thing, and multi-agent systems (MAS) are a perfec...

Jan 20 52 2

English	MMLU (Pass@1)	88.3	87.2	88.5	85.2	**91.8**	90.8
	MMLU-Redux (EM)	88.9	88.0	89.1	86.7	-	**92.9**
	MMLU-Pro (EM)	78.0	72.6	75.9	80.3	-	**84.0**
	DROP (3-shot F1)	88.3	83.7	91.6	83.9	90.2	**92.2**
	IF-Eval (Prompt Strict)	**86.5**	84.3	86.1	84.8	-	83.3
	GPQA Diamond (Pass@1)	65.0	49.9	59.1	60.0	**75.7**	71.5
	SimpleQA (Correct)	28.4	38.2	24.9	7.0	**47.0**	30.1
	FRAMES (Acc.)	72.5	80.5	73.3	76.9	-	**82.5**
	AlpacaEval2.0 (LC-winrate)	52.0	51.1	70.0	57.8	-	**87.6**
	ArenaHard (GPT-4-1106)	85.2	80.4	85.5	92.0	-	**92.3**
Code	LiveCodeBench (Pass@1-COT)	38.9	32.9	36.2	53.8	63.4	**65.9**
	Codeforces (Percentile)	20.3	23.6	58.7	93.4	**96.6**	96.3
	Codeforces (Rating)	717	759	1134	1820	**2061**	2029
	SWE Verified (Resolved)	**50.8**	38.8	42.0	41.6	48.9	49.2
	Aider-Polyglot (Acc.)	45.3	16.0	49.6	32.9	**61.7**	53.3
Math	AIME 2024 (Pass@1)	16.0	9.3	39.2	63.6	79.2	**79.8**
	MATH-500 (Pass@1)	78.3	74.6	90.2	90.0	96.4	**97.3**

Isaak Kamau

A Simple Guide to DeepSeek R1: Architecture, Training, Local...

DeepSeek's Novel Approach to LLM Reasoning

4d ago 706 4

In AI Advances by Wei-Meng Lee

Tips and Tricks for Using Ollama

Learn how to change the models folder, configure and use the REST API, and more

Jan 21 326 1

See more recommendations