DS-GA-1011: Natural Language Processing with Representation Learning, Fall 2024

HW-4: Scaling Language Models and Prompt Engineering

Aditeya Baral
N19186654

Please write down any collaborators, AI tools (ChatGPT, Copliot, codex, etc.), and external resources you used for this assignment here.
Collaborators:
AI tools:
Resources:

By turning in this assignment, I agree by the honor code of the College of Arts and Science at New York University and declare that all of this is my own work.

Acknowledgement: Xiang Pan set up the leaderboard and Lavender Jiang developed the programming question. We drew inspiration from the scratchpad paper by Nye et al. and prompt engineering guide. Ryan Teehan contributed question 2.

Before you get started, please read the Submission section thoroughly.

Submission

Submission is done on Gradescope.

Written: You can either directly type your solution in the released `.tex` file, or write your solution using pen or stylus. A `.pdf` file must be submitted.

Programming: Questions marked with "coding" at the start of the question require a coding part. You should submit `submission.py`.

Due Date: This homework is due on November 13, 2024, at 12 pm Eastern Time.

1 Language Models and Compression

In this problem, we will try to connect language models to lossless compression. The goal of lossless compression is to encode a sequence of symbols $x = (x_1, \ldots, x_n)$ $(x_i \in \mathcal{X})$ following a distribution p to a sequence of bits, i.e. $c : \mathcal{X}^* \to \{0,1\}^*$, such that the original sequence can be recovered from the bit sequence. To increase compression efficiency, we want to minimize the expected number of bits per sequence.
Shannon's source coding theorem states that the minimum number of bits (using any compression algorithm) cannot go below the Shannon entropy of the sequence $H(X) = \mathbb{E}_{x\sim p}[-\log_2 p(x)]$ (note that here we use X to denote the random variable and x to denote the value).

1. [2 points] Now, given a language model q trained on a finite sample of sequences from p. Show that the perplexity of the language model is lower bounded by $2^{H(p)}$:

$$2^{\mathbb{E}_{x\sim p}[-\log_2 q(x)]} \geq 2^{H(p)}$$

[**HINT:** You can use the fact that the KL divergence is non-negative: $D_{\mathrm{KL}}(p\|q) \geq 0$.]

We know that the KL divergence between two distributions p and q is defined by the following,

$$D_{\mathrm{KL}}(p|q) = \mathbb{E}_{x\sim p}\left[\log_2 \frac{p(x)}{q(x)}\right] \tag{1}$$

We're given that $D_{\mathrm{KL}}(p|q) \geq 0$. Thus,

$$\mathbb{E}_{x\sim p}\left[\log_2 \frac{p(x)}{q(x)}\right] \geq 0 \tag{2}$$

$$\mathbb{E}_{x\sim p}[\log_2 p(x) - \log_2 q(x)] \geq 0 \tag{3}$$

$$\mathbb{E}_{x\sim p}[\log_2 p(x)] \geq \mathbb{E}_{x\sim p}[\log_2 q(x)] \tag{4}$$

Multiplying both sides by -1 (which flips the inequality) and simplifying by following the linearity of expectations, we obtain the following,

$$\mathbb{E}_{x\sim p}[-\log_2 p(x)] \leq \mathbb{E}_{x\sim p}[-\log_2 q(x)] \tag{5}$$

The Shannon entropy $H(p)$ is defined by $\mathbb{E}_{x\sim p}[-\log_2 p(x)]$. Thus, we obtain the following,

$$H(p) \leq \mathbb{E}_{x\sim p}[-\log_2 q(x)] \tag{6}$$

We can apply 2^x to both sides. Since 2^x is a monotonically increasing function, it preserves the inequality to obtain the following,

$$2^{H(p)} \leq 2^{\mathbb{E}_{x\sim p}[-\log_2 q(x)]} \tag{7}$$

The right side is the definition of perplexity Perplexity $= 2^{\mathbb{E}_{x\sim p}[-\log_2 q(x)]}$, so by switching sides, we've shown:

$$\text{Perplexity} = 2^{\mathbb{E}_{x\sim p}[-\log_2 q(x)]} \geq 2^{H(p)} \tag{8}$$

2. Recall that in Huffman coding, we construct the Huffman tree based on frequencies of each symbol, and assigning binary codes to each symbol by travesing the tree from the root. For example, given a set of symbols $\{a, b, c\}$ and their corresponding counts $\{5, 10, 2\}$, the codeword for a, b, c are 01, 1, 00, respectively. You may want to review Huffman coding at `https://en.wikipedia.org/wiki/Huffman_coding`.

   Note that instead of using counts/frequencies to construct the tree, we can use any weight proportional to the probability of the symbol, e.g., 5/17, 10/17, 2/17 in the above example. Now we will use a toy corpus to estimate the probabilities of each symbol, derive the Huffman code from the probabilities, and encode a sequence.

   (a) [2 points] Given the following corpus (you can assume each token is separated by a whitespace):

   ```
   the cat was on the mat .
   the cat on the mat has a hat .
   the mat was flat .
   ```

   Estimate the unigram probability of each token in the vocabulary.

We can estimate the unigram probability of each token in the vocabulary by finding the fraction of the total vocabulary it represents such that,

Token	Count	Probability
the	5	$\frac{5}{21} \approx 0.2381$
mat	3	$\frac{3}{21} \approx 0.1429$
.	3	$\frac{3}{21} \approx 0.1429$
cat	2	$\frac{2}{21} \approx 0.0952$
was	2	$\frac{2}{21} \approx 0.0952$
on	2	$\frac{2}{21} \approx 0.0952$
has	1	$\frac{1}{21} \approx 0.0476$
a	1	$\frac{1}{21} \approx 0.0476$
hat	1	$\frac{1}{21} \approx 0.0476$
flat	1	$\frac{1}{21} \approx 0.0476$

Table 1: Unigram probabilities of tokens in the corpus

3. [3 points] Use the above probabilities to construct a Huffman tree for the symbols, and encode the sequence `the mat has a hat` and `the hat has a mat`.

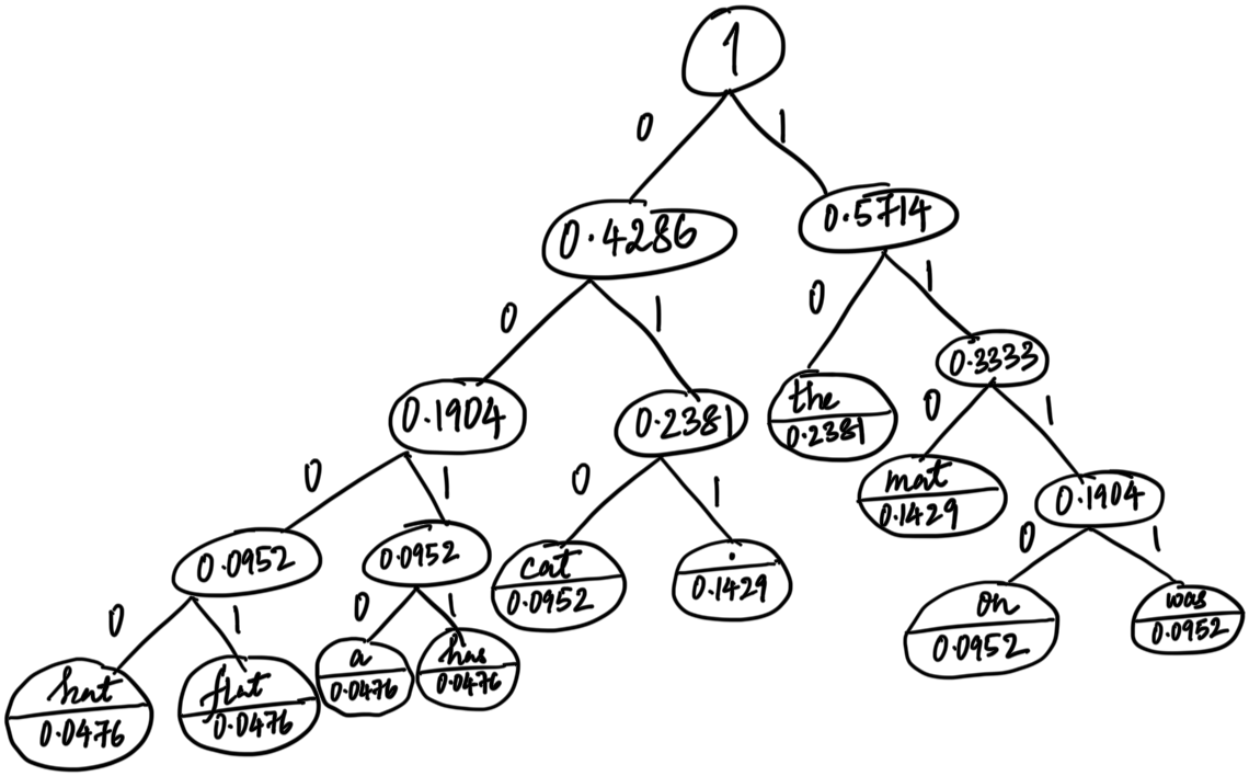

Figure 1: Huffman Tree with Unigram Probabilities

Huffman Encodings:

- `the` → `10`
- `mat` → `110`
- `has` → `0011`
- `a` → `0010`
- `hat` → `0000`

`the mat has a hat` → `10 110 0011 0010 0000`
`the hat has a mat` → `10 0000 0011 0010 110`

4. [4 points] Note that in the above question, the two sequences have the same code length because we are encoding each word independently. In principle, we should be able to shorten the code length by considering dependencies between a word and its prefix. Specifically, given a language model q, for the first word in a sequence, we encode it by constructing a Huffman tree using weights $q(\cdot)$; for the i-th word, we encode it using weights $q(\cdot \mid x_{<i})$. Now, encode the sequence `the mat has a hat` using a bigram language model estimated on the above toy corpus. [**NOTE:** Because we are not using any smoothing, $q(\cdot \mid x_{<i})$ assigns zero probability to certain tokens, which means that we won't be able to encode all possible sequences. But you can ignore this problem in the context of this question.]

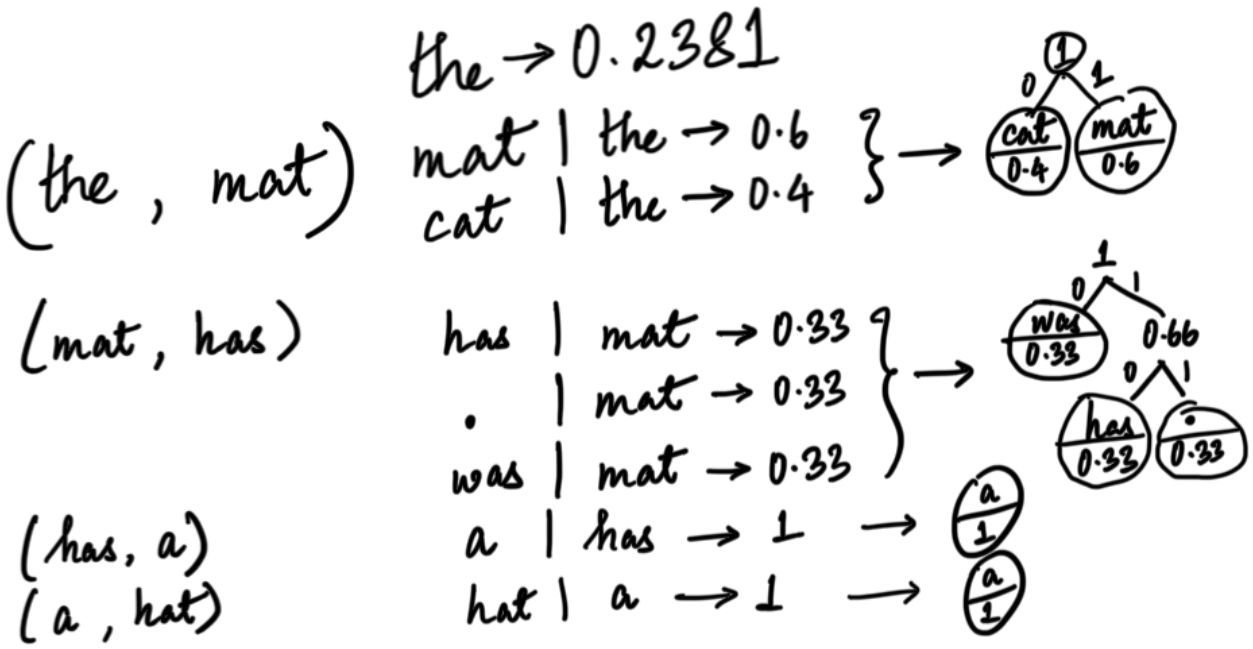

Figure 2: Huffman Tree with Bigram Probabilities

Huffman Encodings:

- `the` → `10`
- `mat` → `1`
- `has` → `10`
- `a` → `0`
- `hat` → `0`

`the mat has a hat` → `10 1 10 0 0`

2 Model Stealing

In this problem, we will use a mathematical treatment of prompting to extract information from a language model. For this problem, assume we have a language model $\mathcal{M}_\theta(p)$, which maps from an input prompt p to a probability distribution $\mathcal{P}(\mathcal{X})$ over a vocabulary $\mathcal{X}$, where $p \in \mathcal{X}^N$ for a prompt of length N. Given a parameterized *trunk* $g_\theta(p)$ which produces the final hidden state, we can write $\mathcal{M}_\theta(p)$ as:

$$\mathcal{M}_\theta(p) = \text{softmax}(\mathbf{W} \cdot g_\theta(p)) \tag{9}$$

where $\mathbf{W} \cdot g_\theta(p) \in \mathbb{R}^\ell$ are the *logits* and $\mathbf{W}$ is the final projection matrix. If the vocabulary size, $|\mathcal{X}|$, is ℓ and the hidden dimension is h, then $\mathbf{W}$ has dimensions $\ell \times h$. Assume that we can directly observe the logits by querying an oracle $\mathcal{O}(p)$, that $h \ll \ell$, and that the weight and activation matrices are full-rank.

1. [2 points] Design an algorithm which allows you to recover the hidden dimension h repeatedly querying the oracle $\mathcal{O}(p)$ for logits. Show that your approach will correctly recover the hidden dimension. [**HINT:** Consider the rank of a matrix of logits]

   (a) We can recover the hidden dimension h from the logits produced by the language model by first generating a set of k diverse prompts $P = p_1, p_2, ..., p_k$, where $k \gg h$ using the information that $h \ll \ell$. These prompts must be diverse enough to potentially span the full hidden space.

   (b) We then query the oracle $\mathcal{O}(p)$ with each prompt p_i in P, obtaining a set of logits $L = \{\mathcal{O}(p_1), \mathcal{O}(p_2), \mathcal{O}(p_k)\}$. Each logit vector $l_i \in \mathbb{R}^\ell = \mathcal{O}(p_i)$ will be a point in the output space corresponding to the hidden state of the model for the respective prompt. The logits matrix L will have dimensions $k \times \ell$.

   (c) We can then compute the rank of this matrix using singular value decomposition (SVD) by counting the number of non-zero singular values to obtain the hidden dimension h. This will decompose matrix L into $U\Sigma V^T$ where Σ contains singular values $\sigma_1 \geq \sigma_2 \geq ... \ \sigma_k \geq 0$. Since the rank is equal to the number of non-zero singular values, we can say the value hidden dimension, h is equal to rank: $h = |i : \sigma_i > 0|$.

   We know this approach will correctly recover the hidden dimension because,

   (a) Each row of L is a logit vector $l_i = \mathbf{W} \cdot g_\theta(p_i)$. The rank of L is limited by:

      i. The rank of $\mathbf{W}$, which is at most h (since $\mathbf{W}$ has h columns). Since $\mathbf{W}$ is assumed to be full-rank, its rank is exactly h.

      ii. The rank of the matrix formed by stacking $g_\theta(p_i)$ for all i.

   (b) If we choose a sufficient number of diverse prompts ($k \gg h$), the hidden states $g_\theta(p_i)$ are likely to span the full h-dimensional space. Since the hidden dimension h constrains the rank of $\mathbf{W} \cdot g_\theta(p)$, the rank of L cannot exceed h, irrespective of the number of prompts k added.

   (c) Therefore, the rank of L will be equal to h, the smaller rank between the rank of $\mathbf{W}$ and rank of the hidden states.

2. [2 points] Building on your previous algorithm, show how we can recover the final projection matrix $\mathbf{W}$ up to linear transformation. That is, show that we can recover $\tilde{\mathbf{W}} = \mathbf{W} \cdot \mathbf{G}$ for some $\mathbf{G} \in \mathbb{R}^{h \times h}$.

We can extend the previous algorithm to recover $\tilde{\mathbf{W}}$ by,

(a) Similar to before, obtain a set of h distinct prompts $P = \{p_1, p_2, ..., p_h\}$. These prompts should be chosen to likely produce linearly independent hidden states.

(b) For each prompt $p_i \in P$, query the oracle $\mathcal{O}(p_i)$ to obtain the corresponding logits vector $l_i \in \mathbb{R}^{\ell}$ and form a matrix L by stacking the logit vectors as columns. We can call the matrix of hidden states $H = [g_\theta(p_1), g_\theta(p_2), \cdots, g_\theta(p_h)]$ and thus express L as,

$$L = \mathbf{W} \cdot [g_\theta(p_1), g_\theta(p_2), \cdots, g_\theta(p_h)] = \mathbf{W} \cdot \mathbf{H} \tag{10}$$

(c) Perform Singular Value Decomposition (SVD) on L such that $L = U\Sigma V^T$ where U is $\ell \times h$, Σ is $h \times h$, and V^T is $h \times h$. We know that $\mathbf{W}$ is full rank and the columns of U form an orthonormal basis for L and the column space of L is the same as $\mathbf{W}$. Since our prompts are assumed to be diverse, $\mathbf{H}$ will be a full-rank and invertible matrix. Thus,

$$L = U\Sigma V^T = \mathbf{W} \cdot \mathbf{H} \tag{11}$$

(d) This implies that U spans the same column space as $\mathbf{W}$ such that,

$$U = \mathbf{W} \cdot \mathbf{G} \tag{12}$$

for some invertible matrix $G \in \mathbb{R}^{h \times h}$.

(e) Therefore, U is $\tilde{\mathbf{W}} = \mathbf{W} \cdot \mathbf{G}$ for some $\mathbf{G} \in \mathbb{R}^{h \times h}$.

3 Prompt Engineering for Addition

The goal of this coding problem is the following:

1. Give you hands-on experience with prompt engineering.
2. Understand the challenges involved in prompt engineering.

Specifically, we will use API from together.ai to teach the LLaMa-2 7B model to add two positive 7-digit integers.
First go through the file `README.md` to set up the environment required for the class.

1. Zero-shot Addition

We provided you a notebook `addition_prompting.ipynb`. In the first section, there are two examples of zero-shot addition: ten 1-digit addition and ten 7-digit additions.

a. (2 points, written) Run the two examples. In your opinion, what are some factors that cause language model performance to deteriorate from 1 digit to 7 digits?

(a) 7-digit addition is more **complex** than single-digit addition since it involves other **arithmetic nuances such as multiple steps and carry-over digits**, which increases the scope for making errors.

(b) Language models are usually trained on text where mostly single or double-digit numbers are common and **7-digit numbers aren't commonly found**. This makes calculations using these numbers difficult as well since it has not seen similar data while training and thus cannot generalise well to higher-order numbers. This disparity leads to better performance on single-digit numbers.

(c) Language models have still not reached the stage where they can perform complex mathematical understanding and reasoning tasks and can generalise mathematical concepts like addition to complex operations. While simpler operations like single-digit addition can be inferred from patterns in text, 7-digit addition is pretty complex and beyond their capabilities.

(d) Addition also includes **positional understanding of numbers and place values**. The way numbers are tokenized also affects their processing and generation. **Large numbers are also tokenised into more tokens**, increasing the input length and potentially leading to **tokens that can distort the calculation**. The model may find it harder to retain and manipulate tokenised representations of large numbers, as opposed to single tokens for 1-digit numbers. This would disrupt not only the model's recognition and understanding of the entire number as a whole but also affect its ability to perform operations accurately since it might split numbers at unexpected positions.

(e) Language models **do not perform explicit numerical computations**. They generate text based on patterns seen in training data. Simple patterns such as single-digit addition can be found and reproduced since these are commonly found in text, but 7-digit addition is complex and more challenging to reproduce by simply replicating a pattern, especially one it has not seen before.

(f) Adding higher-order numbers involves tracking place values and carrying values between digits, which requires **maintaining a long sequence of dependencies**. Language models can struggle with such sequential, memory-intensive operations, as they lack the capacity to keep track of all necessary dependencies accurately over longer sequences.

b. (5 points, written) Play around with the config parameters in together.ai's web UI.

- What does each parameter represent?
- How does increasing each parameter change the generation?

(a) **Output Length**:

- **Represents**: The maximum number of tokens the model will generate given a prompt.
- **Effect of Increasing**: Increases the number of tokens generated, thus leading to longer outputs with more detailed and complete responses.

(b) **Temperature**:

- **Represents**: The creativity and randomness in token selection when tokens are sampled while generating auto-regressively. A lower temperature makes the model more deterministic, while a higher temperature adds variability.
- **Effect of Increasing**: Increases the diversity of tokens generated leading to more creative outputs diverse from the training set, while also increasing the potential for hallucinatory, factually incorrect and less-coherent outputs.

(c) **Top-p**:

- **Represents**: The cumulative probability threshold for token selection while sampling for tokens from a cumulative probability distribution (nucleus sampling). For example, with top-p set to 0.9, the model will only choose from tokens that make up the top 90% probability mass. It adjusts the number of choices for each predicted token, which helps to maintain diversity and generate more fluent and natural-sounding text.
- **Effect of Increasing**: Increases the diversity and variety of tokens generated since it now considers a larger and wider pool of tokens but at the risk of relevancy to the provided prompt.

(d) **Top-k**:

- **Represents**: The maximum number of highest probability tokens to consider. It limits the number of choices for the next predicted word or token to the top k most probable tokens at each step, which helps to speed up the generation process and can improve the quality of the generated text.
- **Effect of Increasing**: Increasing top-k widens the pool of potential tokens increasing the diversity of tokens generated, thus leading to more creative and varied outputs different from the training set but at the risk of coherency or unexpectedness in the output.

(e) **Repetition Penalty**:

- **Represents**: The penalty is applied to discourage the model from repeating tokens or phrases within its response by reducing the probability of already appearing tokens. It also reduces the likelihood of getting stuck in a loop.
- **Effect of Increasing**: Higher repetition penalty reduces redundancy and repetition of tokens in the generated text, leading to more varied, diverse and dynamic responses. It also prevents it from looping through the same phrases, however, it might also force the model to generate uncommon or inapt words to avoid the repetition penalty.

c. (2 points, written) Do 7-digit addition with 70B parameter llama model.

- How does the performance change?
- What are some factors that cause this change?

(a) The performance of the larger language model with a larger number of parameters is **higher** compared to the smaller language model. The **accuracy increases** from `0.1` to `0.8` and the **mean average error (MAE) decreases** from `7774464.2` to `1010.0`.

(b) Factors contributing to this change are,

- **Increased Model Size**
  - The larger parameter count (70B vs 7B) allows the model to **capture more complex patterns and relationships in numerical data** which helps it perform more complex arithmetic additions and replicate patterns for 7-digit addition. While both models are trained on similar data, the larger model can make better use of the same dataset due to its greater parameter count, allowing it to capture more **subtle details and arithmetic patterns** present in the data.
  - With more parameters, the model can also develop a **more nuanced understanding of place value and carry operations in multi-digit arithmetic** which helps it outperform the smaller model.
  - More layers in the 70B model enable **more complex transformations of the input**, potentially allowing for better arithmetic reasoning.
  - With **richer internal representations**, errors in intermediate steps are less likely to compound in larger models thus leading to more accurate reasoning steps and the final answer.
  - Some capabilities, like complex arithmetic, may only emerge at certain model sizes, explaining the significant jump in performance.
- **Improved Context Handling**
  - Larger models tend to be better able to **maintain and utilise longer context**, which is crucial for multi-step calculations like 7-digit addition.
  - A larger model can **retain and manipulate information over longer dependencies** more effectively. In addition, tasks like multi-digit addition require tracking a sequence of steps such as summing each digit column and managing carry values. The 70B model's larger parameter count helps maintain and utilise this context more reliably, improving accuracy.
- **Increased Capacity for Pattern Recognition**: With more parameters, the 70B model has a greater capacity to **learn complex patterns and relationships**, such as carrying operations and multi-digit place values in arithmetic tasks. This makes it **better equipped to recognise patterns** in 7-digit numbers, which are more complex and nuanced than single-digit calculations.
- **Improved Generalisation**: The larger model size helps it **generalise arithmetic concepts more effectively**, even for tasks it was not explicitly trained on, due to its **enhanced ability to learn from the patterns in the data**. This results in fewer errors, reflected in the lower mean average error (MAE).

d. (2 points, written) Previously we gave our language model the prior that the sum of two 7-digit numbers must have a maximum of 8 digits (by setting `max_token=8`). What if we remove this prior by increasing the `max_token` to 20?

- Does the model still perform well?
- What are some reasons why?

(a) The performance of the model drops. The **accuracy decreases** to `0.1` and the **mean average error (MAE) significantly increases** to `2665190330412275.0`.

(b) Factors contributing to this change are,

- **Lack of Constrained Search Space**
  - Setting `max_token=8` provided a useful constraint that matched the mathematical reality that the sum of two 7-digit numbers cannot exceed 8 digits. With a larger `max_token`, the model has **more freedom to generate outputs with additional, unnecessary digits**, leading to numbers that are mathematically impossible for 7-digit addition, potentially leading to more errors or nonsensical outputs and a sharp drop in accuracy.
  - With more freedom in output length, the model is **more prone to hallucinating** or generating fictitious large numbers as well as other incoherent or irrelevant text.
  - Models often rely on implicit priors and constraints in response length when performing specific tasks. Removing this prior by increasing the `max_token` size means the model can **no longer rely on the assumption that the sum of two 7-digit numbers should have at most 8 digits**. This affects the model's ability to align the output format with the expected answer.
  - This lack of guidance increases the likelihood of large errors, as indicated by the significant rise in mean average error (MAE).
- **Increased Complexity**
  - A larger output space makes the task **more complex**, affecting the model's ability to maintain coherence in longer sequences.
  - With more tokens available, **errors in the model's arithmetic reasoning can compound** and lead to significantly larger numerical errors.
  - Allowing up to 20 tokens might cause the model to **deviate from the desired answer format** and produce extraneous tokens, as it has more freedom in token generation.

2. In Context Learning

In this part We will try to improve the performance of 7-digit addition via in-context learning. For cost-control purposes (you only have $25 free credits), we will use llama-2-7b.

a. (2 points, written) Using the baseline prompt ("Question: What is 3+7? Answer: 10 Question: What is a+b? Answer:"), check 7-digit addition for 10 pairs again.

- Compared to zero-shot 7-digit additions with maximum 7 tokens, how does the performance change when we use the baseline in-context learning prompt?
- What are some factors that cause this change?

(a) The performance on the baseline in-context learning prompt is **similar** to the performance of the language model with zero-shot prompting. Its **accuracy** is `0.0` and **mean average error (MAE)** is `1261435.0`. However, we do notice a decrease in the MAE score of the model with in-context learning, thus hinting that it has generated answers closer to the original answer and adding an example to the prompt helps the model replicate patterns better. However, there is much scope for improvement since the overall accuracy is still zero.

(b) Factors contributing to this change are,

- **Inadequate Example Complexity**
  - The baseline prompt provides a very simple example (`3+7 = 10`) that **doesn't adequately represent the complexity of 7-digit addition nor provide specific guidance** for handling large numbers or carry operations.
  - The jump from single-digit addition to 7-digit addition is **too large for the model to generalize** effectively, leading to a lack of improvement over zero-shot performance.
- **Insufficient Context for Pattern Induction**
  - Language models rely heavily on pattern recognition in in-context learning. A single and simple example **fails to provide a sufficient pattern** for the model to generalize 7-digit addition. To be effective, a **prompt should include examples similar to the task** at hand such as having 7-digit addition examples.
  - The format of the example does not represent the 7-digit addition task since it **missing crucial steps** and thus doesn't demonstrate the process of solving multi-digit addition problems. An ideal prompt would include **steps on how to add the digits** one place at a time.
  - Without exposure to examples demonstrating complex mathematical operations like **carry-overs and multi-step addition**, the model is unlikely to perform these operations correctly.
- **Smaller Model Size**
  - Being a **smaller model** compared to the 70B variant, it might not have the capacity to perform complex arithmetic accurately without more extensive prompting. The model could have performed better if it had more parameters to understand and replicate the pattern and generalise to 7-digit addition.
  - It thus **fails to transfer** the simple addition concept to more complex arithmetic operations.

b. (3 points, written) Now we will remove the prior on output length and re-evaluate the performance of our baseline one-shot learning prompt. We need to modify our post processing function to extract the answer from the output sequence.

- Describe an approach to modify the post processing function.
- Compared to 2a, How does the performance change when we relax the output length constraint?
- What are some factors that cause this change?

(a) We can modify the post-processing function by,

- **Extracting the first valid sequence of digits**, stopping at the first non-digit character or end of the string instead of only the first line. This would ensure we **capture the result even if the model's answer spans multiple lines or contains additional irrelevant text**. It would return the intended number (if present) while ignoring any extra content that may have been generated.
- **Match with patterns** such as `"Answer: [number]"` instead of just looking for digits since this would ensure **we only pick up the answer present among various other numbers** in the generated response.
- We could also potentially search for all numbers in the entire output and **return the largest one** since it is likely to be the final sum in an addition problem.

(b) The performance with a relaxed output length constraint is **similar** to the performance of the language model with a fixed output length. Its **accuracy** is `0.0` and **mean average error (MAE)** is `10395014.6`. However, we notice a **decrease in the MAE** metric, indicating it has generated answers **closer** to the original answer.

(c) Factors contributing to this change are,

- Without length constraints, the model can generate larger arithmetic sequences corresponding to arithmetic operations with larger numbers **without truncating or tokenising digits** or worrying about the max length, which potentially might lead to correct answers.
- Removing the maximum token constraint gives the model **more freedom to generate longer answers**, which may capture more digits correctly, leading to a reduced MAE.
- This flexibility allows the model to produce outputs that are **closer to the correct magnitude**, even if the exact answer isn't achieved.
- Without length constraints, the model is **less likely to cut off digits** at the end of large sums.
- Allowing the model to generate text beyond the final answer might assist it in potentially reaching the final answer since it **no longer is forced to generate only the right answer** but rather can sequentially generate the most probable answer based on the pattern in the prompt.
- Longer outputs allow the model to **consider multiple potential answers at every step while sampling tokens**, thus having the chance to leverage more of its internal knowledge about arithmetic operations with longer outputs and increasing the chance of including a closer approximation.

c. (4 points, written) Let's change our one-shot learning example to something more "in-distribution". Previously we were using 1-digit addition as an example.
Let's change it to 7-digit addition (`1234567+1234567=2469134`).

- Evaluate the performance with max_tokens = 8. Report the `res, acc, mae, prompt_length`.
- Evaluate the performance with max_tokens = 50. Report the `res, acc, mae, prompt_length`.
- How does the performance change from 1-digit example to 7-digit example?
- Take a closer look at `test_range`. How was `res` calculated? What is its range? Does it make sense to you? Why or why not?

(a) **Performance with `max_tokens = 8`**

- `res = 0.000369`
- `acc = 0.2`
- `mae = 4280824.9`
- `prompt_length = 78`

(b) **Performance with `max_tokens = 50`**

- `res = 0.000232`
- `acc = 0.1`
- `mae = 4093593.2`
- `prompt_length = 78`

(c) **Performance comparison with `max_tokens = 8`**

- **1-digit Example Performance**: `acc = 0.0, mae = 1261435.0`
- **7-digit Example Performance**: `acc = 0.2 [0.2 increase], mae = 4280824.9 [239% increase]`.

Performance comparison with `max_tokens = 50`

- **1-digit Example Performance**: `acc = 0.0, mae = 10395014.6`
- **7-digit Example Performance**: `acc = 0.1 [0.1 increase], mae = 4093593.2 [60.6% decrease]`

(d) `res` is calculated by multiplying the accuracy score `acc` with the inverse prompt length `prompt_length` and then scaling it with the complement of the mean absolute error `mae` which is in turn scaled down by a numerical factor of 5×10^6. That is,

$$\texttt{res} = \texttt{acc} \times \frac{1}{\texttt{prompt\_length}} \times \left(1 - \frac{\texttt{mae}}{5 \times 10^6}\right) \tag{13}$$

The range of `acc` is `[0, 1]` since it is a fraction of the total examples whose sum has been correctly computed and the range of $\frac{1}{\texttt{prompt\_length}}$ is `(0, 1]` since `prompt_length` is always $1 <= \texttt{prompt\_length} < \infty$. The range of the third term $(1 - \frac{\texttt{mae}}{5 \times 10^6})$ will return negative values when $\texttt{mae} > 5 \times 10^6$, but will return positive values if $\texttt{mae} < 5 \times 10^6$, thus its range is $(-\infty, 1]$. Thus, the theoretical range of `res` is lower bound to negative values (when $\texttt{mae} > 5 \times 10^6$) and upper bound to 1 when `acc = 1`, `prompt_length = 1` and `mae = 0`.

$$\texttt{res} \in (-\infty, 1] \tag{14}$$

Although the metric combines multiple factors, it has a few limitations being unreliable and difficult to interpret.

- **Unsuitable Range of Values**: The MAE for 7-digit addition can easily exceed 5×10^6, making the overall term negative, which doesn't make sense for an error metric.

- **Lack of Interpretability**: The result doesn't have a clear meaning or scale and isn't very easy to interpret because it combines metrics which might not necessarily combine logically. It is not directly interpretable as a standard performance score (like accuracy or MAE alone), so it might not be as informative as these individual components. Additionally, the 5×10^6 factor seems arbitrarily chosen and may not be appropriate for all ranges of addition problems.
- **Dependency on Prompt Length**: Shorter prompts are favoured, which may not always correlate with better performance. This makes sense if prompt efficiency is crucial, but it may lead to poor results if shorter prompts lead to poorer performance.
- **Emphasis on accuracy**: Small changes in accuracy can overshadow large changes in MAE.

To improve the metric, we could use accuracy and MAE separately or develop a more carefully scaled combined metric by using a revised scaling factor for MAE and weigh prompt length differently.

d. (written, 4 points) Let's look at a specific example with large absolute error.

- Run the cell at least 5 times. Does the error change with each time? Why?
- Can you think of a prompt to reduce the error?
- Why do you think it would work?
- Does it work in practice? Why or why not?

(a) **Yes, the mean average error (MAE) changes every time with each run of the code cell**. This is because with each run, **a different answer is generated** for the question `"9090909 + 1010101?"` due to the probabilistic nature of the model. Since the absolute error depends on the answer being generated, the magnitude of the **error also changes with each run** of the code cell. However, since all runs of the code cell produce an incorrect answer, the **final accuracy always remains** `0.0`, regardless of the answer being generated.

(b) We can reduce the error by providing a more effective prompt that offers an example of 7-digit addition and a **step-by-step calculation of how to proceed with multi-digit addition and handling place values with carryovers**. An example of such a prompt is,

```
"Question:  What is 1234567+1354634 ?
Answer:  Start from the rightmost digit and remember to carry over when
necessary = 2589201
Question:  What is "
```

(c) This prompt is likely to work because,

- This prompt provides the model with **specific guidance on performing multi-digit addition** with a clear structure for the addition task, including an example with a similar structure and the carry operation.
- These cues will help the model **identify the pattern of summing columns** by treating each digit independently and summing them from the least significant place to the most significant place and carrying over, which is essential for accurate large-digit arithmetic. By including a carryover reminder, the prompt **encourages the model to handle cases where the sum of digits exceeds 9**, making it more likely to generate accurate results.
- By showing an example that **matches the problem's structure and complexity** and ***how* to solve** it, the prompt aims to steer the model toward correct calculations and **primes it to approach the subsequent question in a similar, structured manner**. It also helps set an expectation for the output format and answer structure.
- The fixed output pattern encourages the model to apply the same addition pattern when generating responses, thereby **reducing random variability** in its answers.

(d) Yes, the prompt works in practice. On an average over > 5 runs, the prompt **produced an average absolute error of** `1009.0`, including an **absolute error of** `0.0` **in one run**, indicating that it was **able to generate the correct answer once**. Although it generated the incorrect answer on all runs but one, it produced **significantly lower mean average error scores** compared to the baseline prompt. The prompt worked because it provided specific guidance to the model to solve the problem at hand by,

- It encouraged a **step-by-step approach**, mimicking human addition methods. This structured method helps break down complex tasks into manageable steps, improving the model's ability to handle multi-digit addition.
- It reminds the model to **pay attention to each digit** and the **carry-over process**. By providing clear instructions like `"Start from the rightmost digit"` and `"remember to carry over when necessary"` the prompt guided the model through the critical steps of the addition process. The relevant example (which included a carryover) reinforced this key arithmetic concept, which is especially critical for large-number addition. Without these hints, the model would be more prone to make errors when a carry-over was necessary.

- The detailed prompt **reduced the range of possible responses** by guiding the model toward answers similar in format to the example provided. This led to more consistent results and reduced the occurrence of extreme errors that might arise from the model attempting unconventional approaches.
- The example **included numbers of the same length** as the target question, which helped the model to generalise and apply the same method to the actual problem. Such targeted examples make it more likely that the model will pick up on the patterns necessary for accurate large-digit addition.
- It provides a **clear structure** for the addition task. The instruction likely helped the **model understand that it involved place-by-place addition** rather than simple pattern matching.

3. Prompt-a-thon!

In this part, you will compete with your classmates to see who is best at teach llama to add 7-digit numbers reliably! Submit[1] your `submission.py` to enter the leader board!

The autograder will test your prompt on 30 pairs of 7-digit integer addition. Some pairs are manually designed and some pairs are randomly generated using a random seed that is different from `addition_prompting.ipynb`. This will generate 30 API calls, with 1 second wait time between each API call. Since prompting has randomness, we will run the same trial 3 times and report the average performance. In total, each submission will incur 90 API calls.

Please keep `max_tokens` to at least 50, so that you're not giving the model a prior on the output length. Please also do not try to hack the pre-processing and post-processing functions by including arithmetic operations. The autograder is able to detect the use of arithmetic operations in these functions.
Here's a list of functions you can modify:

- `your_api_key`
- `your_prompt`
- `your_config`
- `your_pre_processing` (but no arithmetic operation. e.g., `f"a+b={a+b}"` is not allowed.)
- `your_post_processing` (but no arithmetic operation. e.g., `f"a+b={a+b}"` is not allowed.)

a. (coding, 5 points) Use prompt to improve test performance. For full credit, either

   - get average accuracy greater than 0.1, or
   - get average mean absolute error less than 5e6.

   To prevent recovery of test pairs, your autograder submission is limited to 10 tries. For testing, we recommend using `test_prompts.py`.

b. (optional, 1-3 points) Top 3 students on the leaderboard will be given the following awards:

   (a) First place: 3 points
   (b) Second place: 2 points
   (c) Third place: 1 point

   We will add the points to your homework 4 score (capping at the maximum). For example, suppose the student in first place originally has 23 out of 25, with the reward they will get 25 out of 25. Suppose the student in first place originally has 20 out of 25, with the reward they will get 23 out of 25. In case of a tie, we recognize the earlier submission.

[1] Note: while you can use `prompt.txt` for debugging and local testing, for the final autograder submission, please use string (**not file**). The reason why is that autograder cannot find `prompt.txt` in the testing environment. Sorry about the inconvenience!