

# LLM Reasoning

## Lecture 2: Repeated Sampling

Yaya Sy

12/10/2025

# Recap on LLM Sampling

→ The **LLM learns a probability function** by **maximizing the likelihood** of the training data (the Web)

→ After training, we can **generate text by sampling** one token at a time in the learned probability distribution

→ There are many sampling methods, and it's **a trade-off between diversity and coherence**

- *Greedy*: predict the token with the highest score
- *TopK*: sample within the  $k$  tokens with the highest scores
- *Temperature*: redistribute the probability distribution to give some tokens more chance to be sampled
- *TopP*: samples only from tokens that concentrate the probability mass

# The limits of single sampling

Sampling is important because, for now, it's the only way to ***elicit*** the capabilities encoded within the LLM.

But sampling only one answer per question is not sufficient

The LLM needs more ***time***, more ***attempts***.

# The limits of single sampling

The case where the LLM samples only a single answer

$LLM(\dots | c = \text{Is Ronaldo better than Messi?}) \rightarrow$

What if the LLM has more attempts?

$LLM(\dots | c = \text{Is Ronaldo better than Messi?}) \rightarrow \left\{ \right.$

# The limits of single sampling

The case where the LLM samples only a single answer

$LLM(\dots | c = \text{Is Ronaldo better than Messi?}) \rightarrow \text{I prefer cats}$

What if the LLM has more attempts?

$LLM(\dots | c = \text{Is Ronaldo better than Messi?}) \rightarrow \left\{ \right.$

# The limits of single sampling

The case where the LLM samples only a single answer

$LLM(\dots \mid c = \text{Is Ronaldo better than Messi?}) \rightarrow \text{I prefer cats}$

What if the LLM has more attempts?

$LLM(\dots \mid c = \text{Is Ronaldo better than Messi?}) \rightarrow \left\{ \begin{array}{l} \text{I love Yamal's haircut} \end{array} \right.$

# The limits of single sampling

The case where the LLM samples only a single answer

$LLM(\dots | c = \text{Is Ronaldo better than Messi?}) \rightarrow \text{I prefer cats}$

What if the LLM has more attempts?

$LLM(\dots | c = \text{Is Ronaldo better than Messi?}) \rightarrow \begin{cases} \text{I love Yamal's haircut} \\ \text{Ronaldo is better} \end{cases}$

# The limits of single sampling

The case where the LLM samples only a single answer

$LLM(\dots | c = \text{Is Ronaldo better than Messi?}) \rightarrow \text{I prefer cats}$

What if the LLM has more attempts?

$LLM(\dots | c = \text{Is Ronaldo better than Messi?}) \rightarrow \begin{cases} \text{I love Yamal's haircut} \\ \text{Ronaldo is better} \\ \dots \\ \text{Messi is better} \end{cases}$



# The limits of single sampling

The case where the LLM samples only a single answer

$LLM(\dots | c = \text{Is Ronaldo better than Messi?}) \rightarrow \text{I prefer cats}$

What if the LLM has more attempts?

$LLM(\dots | c = \text{Is Ronaldo better than Messi?}) \rightarrow \begin{cases} \text{I love Yamal's haircut} \\ \text{Ronaldo is better} \\ \dots \\ \text{Messi is better} \end{cases}$

# The limits of single sampling

**Strong hypothesis:** LLMs can solve any task in the world if:

- we have the ***budget*** to ***scale infinitely the number of attempts***
- we have a way to ***pick*** the best answer among the generated answers

***Proof:*** *Infinite monkey theorem*

# Infinite monkey theorem

---

[Article](#) [Talk](#)

---

From Wikipedia, the free encyclopedia

The **infinite monkey theorem** states that a [monkey](#) hitting keys independently and at [random](#) on a [typewriter](#) keyboard for an [infinite](#) amount of time will [almost surely](#) type any given text, including the complete works of [William Shakespeare](#).

# Infinite monkey theorem

---

[Article](#) [Talk](#)

---

From Wikipedia, the free encyclopedia

The **infinite monkey theorem** states that a [monkey](#) hitting keys independently and at [random](#) on a [typewriter](#) keyboard for an [infinite](#) amount of time will [almost surely](#) type any given text, including the complete works of [William Shakespeare](#).



The monkey becoming *Shakespeare* after the  $\infty$ -th attempt

# Infinite monkey theorem

[Article](#) [Talk](#)

From Wikipedia, the free encyclopedia

The **infinite monkey theorem** states that a [monkey](#) hitting keys independently and at [random](#) on a [typewriter](#) keyboard for an [infinite](#) amount of time will [almost surely](#) type any given text, including the complete works of [William Shakespeare](#).

Are LlaMAs (=LLMs) just monkeys?



The monkey becoming *Shakespeare* after the  $\infty$ -th attempt

# Infinite monkey theorem

[Article](#) [Talk](#)

From Wikipedia, the free encyclopedia

The **infinite monkey theorem** states that a [monkey](#) hitting keys independently and at [random](#) on a [typewriter](#) keyboard for an [infinite](#) amount of time will [almost surely](#) type any given text, including the complete works of [William Shakespeare](#).

Are LlaMAs (=LLMs) just monkeys?



Your Llama becoming the best French *translator* after the  $\infty$ -th attempt

# LlaMAs (=LLMs) are maybe just monkeys

User: Using the numbers [19, 36, 55, 7], create an equation that equals 65.

Assistant: Let me solve this step by step.

<think> We need to find an equation using the numbers 19, 36, 55, and 7 exactly once, with basic arithmetic operations, that equals 65. One possible combination is  $55 + 36 - 19 + 7$ . Let's check:  $55 + 36 = 91$ ,  $91 - 19 = 72$ , and  $72 + 7 = 79$ . However, this doesn't equal 65. Another combination is  $55 + 36 + 7 - 19$ . Let's check:  $55 + 36 = 91$ ,  $91 + 7 = 98$ , and  $98 - 19 = 79$ . This also doesn't equal 65. After trying different combinations, I found that  $55 + 36 + 7 - 19 = 79$ , which is close but not equal to 65. Finally, I found that  $55 + 36 - 19 + 7 = 79$ , which is still not equal to 65. However, if we try  $55 + 36 - 7 + 19$ , we get  $55 + 36 = 91$ ,  $91 - 7 = 84$ , and  $84 + 19 = 103$ , which is not equal to 65. After further trial and error, I found that  $55 + 36 + 7 - 19 = 79$ , which is still not equal to 65. However, if we try  $55 + 36 - 7 - 19$ , we get  $55 + 36 = 91$ ,  $91 - 7 = 84$ , and  $84 - 19 = 65$ . This equals 65. </think>

<answer>  $55 + 36 - 7 - 19$  </answer>

Looks more as brute force than *intelligent* search



# LlaMAs (=LLMs) are probably better than monkeys

User: Using the numbers [19, 36, 55, 7], create an equation that equals 65.

Assistant: Let me solve this step by step.

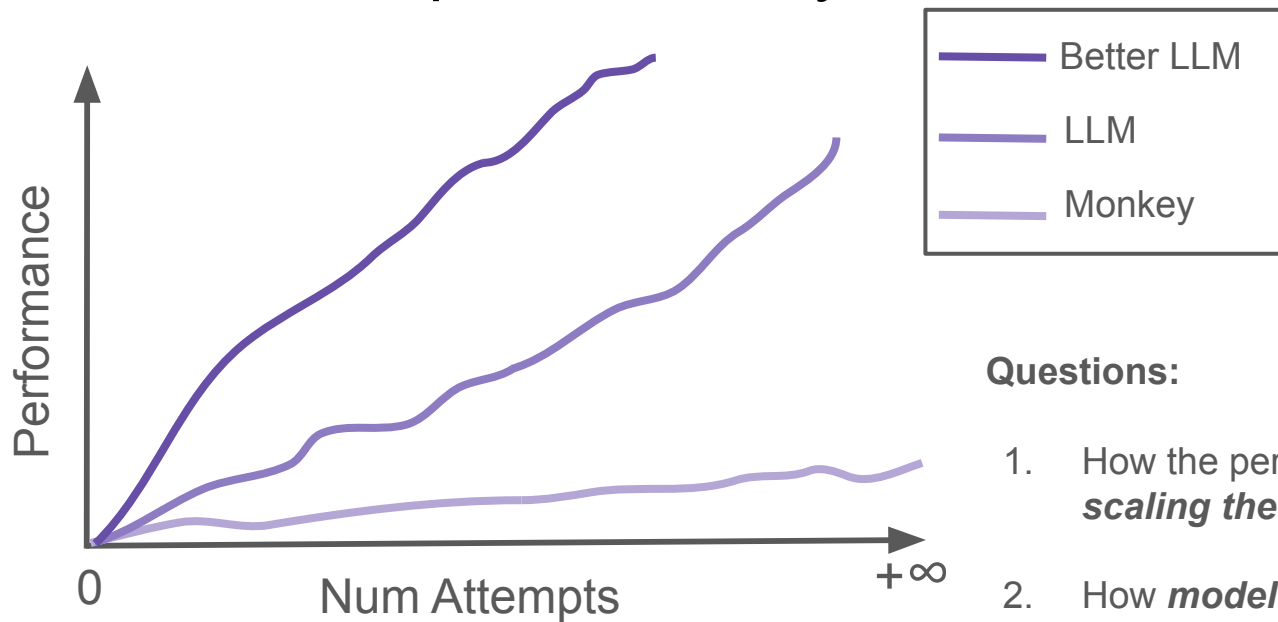
<think> We need to find an equation using the numbers 19, 36, 55, and 7 exactly once, with basic arithmetic operations, that equals 65. One possible combination is `watermelon`. Let's check:  $55 + @j)al!0\# = 72$ , and  $72 + 7 = 79$ . However, this doesn't equal 65. Another combination is  $55 + 36 + 7 - 19$ . Let's check: `zumba` = 98, and  $98 - 19 = 79$ . This also doesn't equal 65. After trying different combinations, I found that  $55 + 36 + 7 - 19 = 79$ , which is close but not equal to 65. Finally, I found that  $55 + 36 - 19 + 7 = 79$ , which is still not equal to 65. However, if we try  $55 + 36 - 7 + 19$ , we get  $55 + 36 = 91$ ,  $91 - 7 = 84$ , and  $84 + 19 = 103$ , which is not equal to 65. After further trial and error, I found that  $55 + 36 + 7 - 19 = 79$ , which is still not equal to 65. However, if we try  $55 + 36 - 7 - 19$ , we get  $55 + 36 = 91$ ,  $91 - 7 = 84$ , and  $84 - 19 = 65$ . This equals 65. </think>

<answer>  $55 + 36 - 7 - 19$  </answer>

LLMs are better than monkeys: it's not totally random



# More or less capable monkeys



## Questions:

1. How the performance evolves when ***scaling the number of attempts***?
2. How ***model's capability*** affects the number of attempts and performance?
3. What about ***costs***?

# Repeated Sampling in a Nutshell

In math:

$$\hat{y} = \arg \max_{y \in p(\dots | c, \theta)} R(y)$$

$R(y)$  or *reward*( $y$ ) is the function that assigns higher **score** to better answers. The reward can also be **binary** (correct or not)

In pseudo-code:

```
best_score = -Inf
best_candidate = ""
for i in range(num_attempts):
    candidate = sample ~ p(...|c)
    score = reward(candidate)
    if score > best_score:
        best_score = score
        best_candidate = candidate
```

Questions:

1. How the performance evolves when **scaling the number of attempts**?
2. How **model's capability** affects the number of attempts and performance?
3. What about **costs**?

# Scaling Study for Repeated Sampling

<https://arxiv.org/pdf/2407.21787>

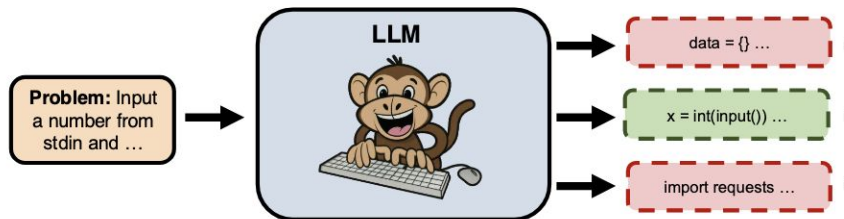
# Verifiable tasks the LLM must solve

**Verifiable task:** it's possible to say *objectively* if the solution to the task is correct or not.

- **GSM8K:** A dataset consisting of grade-school level math word problems
- **MATH:** A collection of math word problems that are generally more difficult than those found in GSM8K
- **MiniF2F-MATH:** A dataset of mathematics problems that have been formalized for use with proof-checking languages.
- **CodeContests:** A set of competitive programming problems that includes text descriptions and hidden input-output test cases to verify solution correctness.
- **SWE-bench Lite:** A dataset of real-world GitHub issues must solve by editing the codebase, verified by existing unit tests.

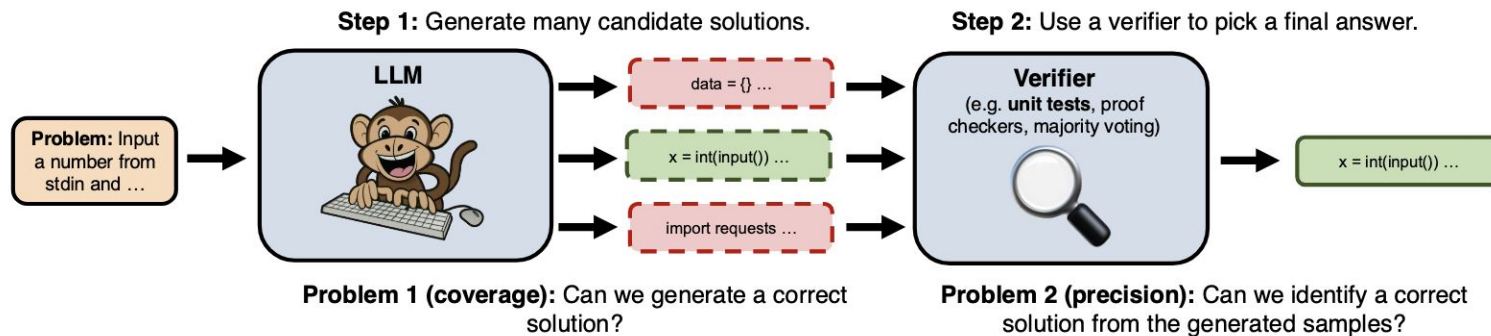
# Scaling Repeated Sampling

**Step 1:** Generate many candidate solutions.

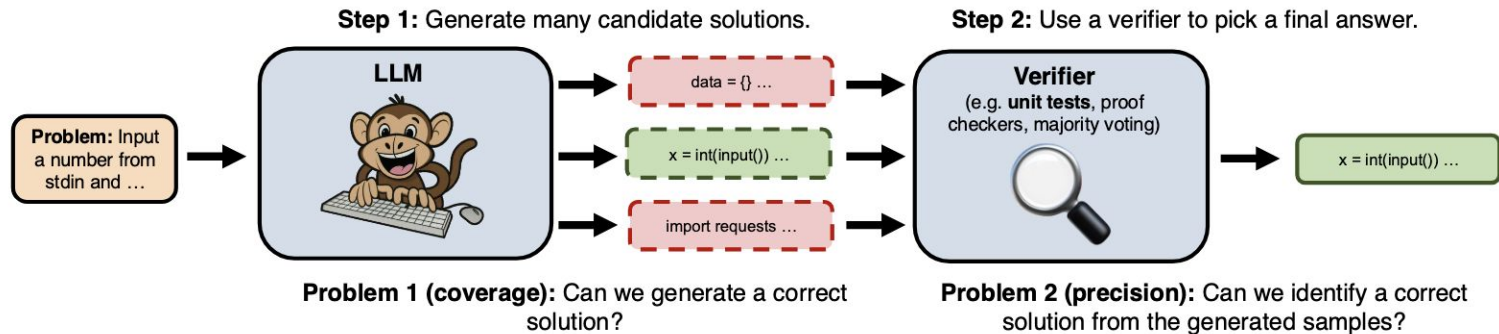


**Problem 1 (coverage):** Can we generate a correct solution?

# Scaling Repeated Sampling



# Scaling Repeated Sampling

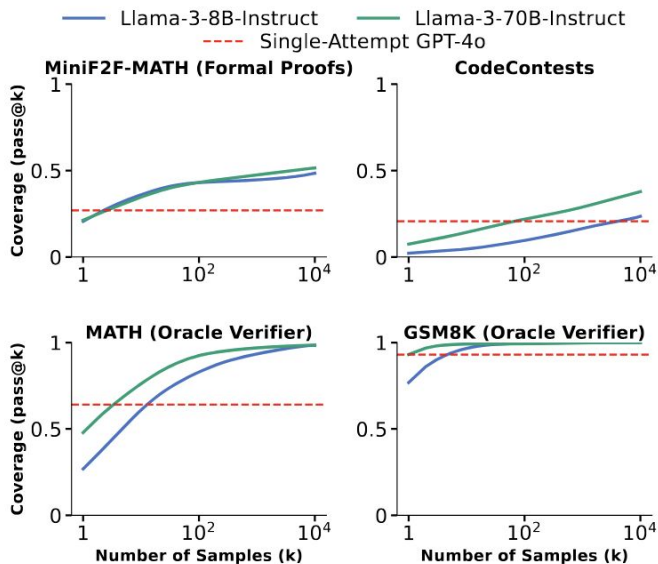


## Questions:

1. How the performance evolves when *scaling the number of attempts*?
2. How *model's capability* affects the number of attempts and performance?
3. What about *costs*?

# Repeated Sampling vs Performance

**Coverage:** % of problems where the LLM generated the right answer at least once



→ When  $k=1$  (single-attempt), GPT-4o is better

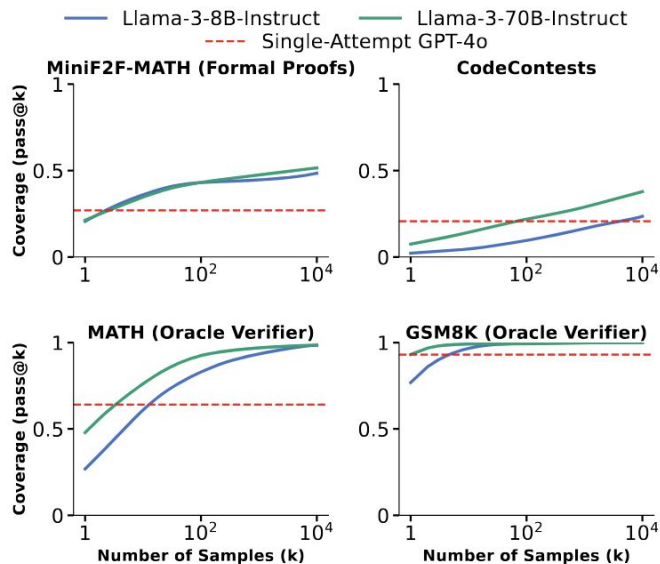
→ Improving the number of attempts (samples) improves performance compared to single-attempt

→ The performance doesn't improve linearly



# Repeated Sampling vs Performance

**Coverage:** % of problems where the LLM generated the right answer at least once



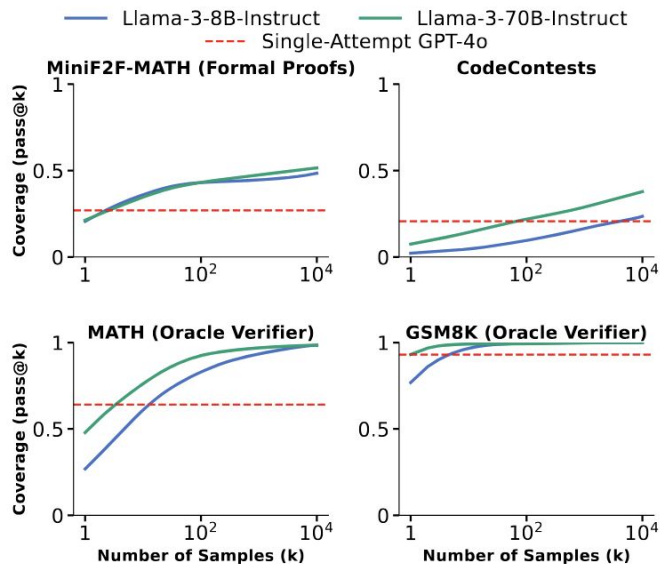
→ When  $k=1$  (single-attempt), GPT-4o is better

→ Improving the number of attempts (samples) improves performance compared to single-attempt

→ The performance doesn't improve linearly

# Repeated Sampling vs Performance

**Coverage:** % of problems where the LLM generated the right answer at least once



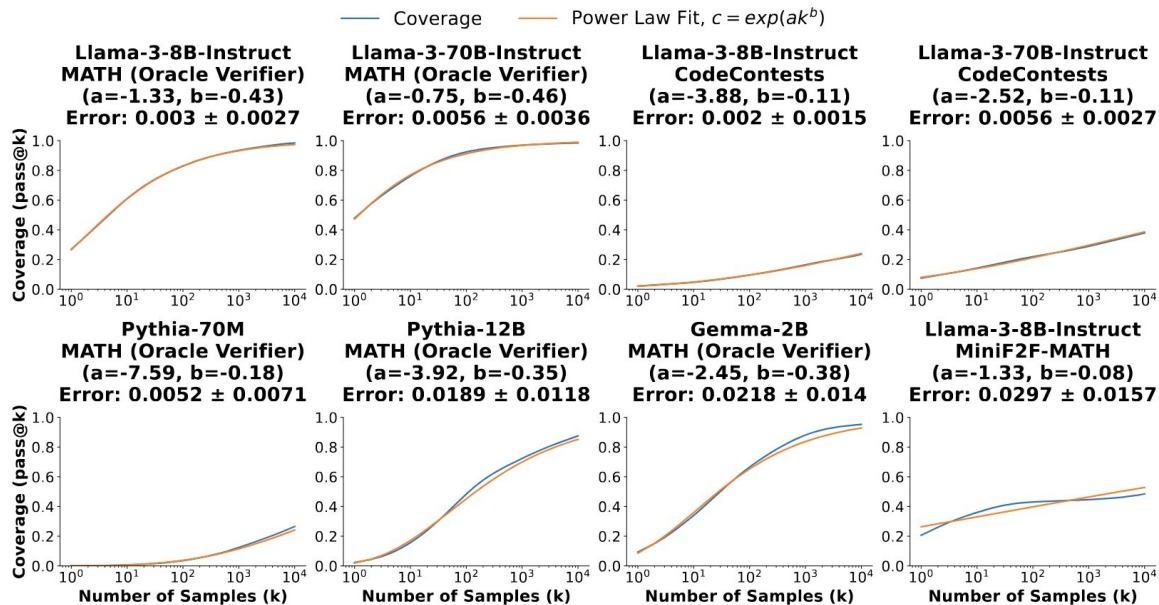
→ The performance doesn't improve linearly

→ We want a function that can predict the performance (coverage) for a given  $k$  (attempts):

$$c = f(k)$$

→  $f$  is the scaling law of performance as a function number of attempts

# Repeated Sampling vs Performance

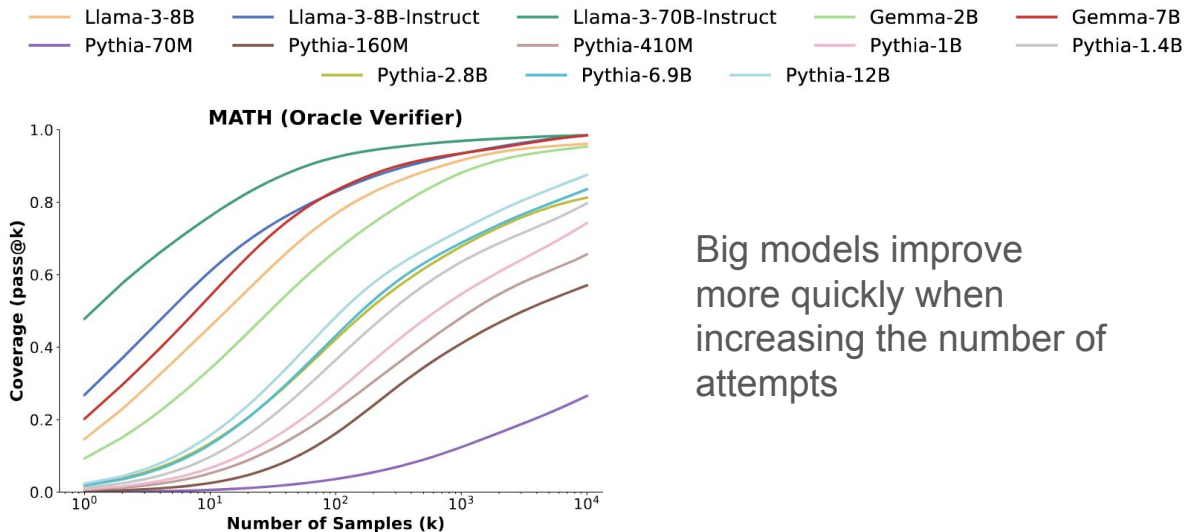


$$c = f(k)$$

$$f(k) = \exp(ak^b)$$

A Power Law governs the relation between the number of attempts  $k$  and the performance (coverage)

# Repeated Sampling vs Performance vs Model Sizes



Big models improve more quickly when increasing the number of attempts

$$c = f(k)$$

$$f(k) = \exp(ak^b)$$

A Power Law governs the relation between the number of attempts  $k$  and the performance (coverage)

# Repeated Sampling vs Performance vs Costs

Model	Cost per attempt (USD)	Number of attempts	Issues solved (%)	Total cost (USD)	Relative total cost
DeepSeek-Coder-V2-Instruct	0.0072	5	29.62	10.8	1x
GPT-4o	0.13	1	24.00	39	3.6x
Claude 3.5 Sonnet	0.17	1	26.70	51	4.7x

It is sometimes more cheap to do repeated sampling using a weak small model than single sampling using a bigger model.

# Summary

- Sampling a single answer for a query can be insufficient
- Generating multiple candidate answers improves the performance
  - Power Law between performance and number of attempts
  - Big models improve more quickly when scaling the number of attempts
  - Doing multiple attempts using a small cheap model can be more cost efficient than doing a single-attempt using a big model

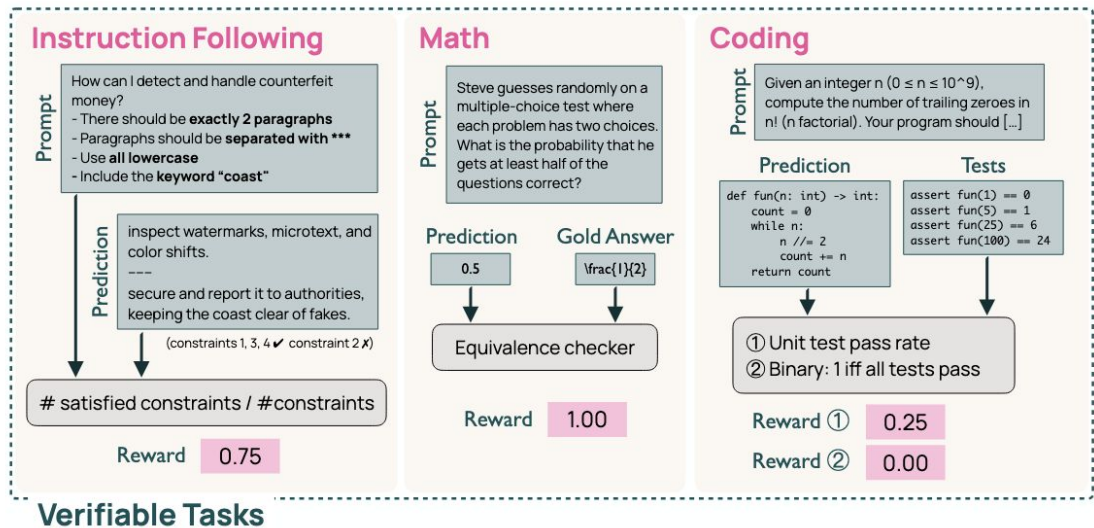
On the Reward

# On the Reward

- The LLM samples multiple output candidates for a prompt  $c$ :  $y \sim \text{sample LLM}(\cdot|c)$
- A reward function  $R(y)$  scores each output  $y$ , and the best is picked
- How to define the reward function?
- This depends on your task
- Some tasks are more easy to verify than others



# Things easy to verify



*Paper: Olmo 3.*

**Instruction Following:** The LLM learns to follow a set of constraints (synthetic or not). The task is verifiable: just use the % of constraints satisfied as reward

**Math.** The task is verifiable: just check against a calculator, proof checker, etc. for rewarding.

**Code.** The task is verifiable: just check against unit tests or run it and observe the result.

# Things not easy to verify

→ **Translation:** There is no way to check the correctness of a translation. There is not only one good translation. Some translations can be better than others.

→ **Creative writing:** This is culture dependant. There is no one ***objective*** way to say a text is more creative than another. It depends on the person, the historical and social context.

→ **Humor:** You can laugh alone and others will think you're crazy. Why do you find it funny?

→ **Do you have any other example?**

# Self-Confidence for non-verifiable tasks

- The ***confidence*** of the LLM about its own answer. A low ***entropy*** means the LLM is more confidence.
- The ***probability*** of the answer computed using the LLM itself can also be used

# Self-Confidence for non-verifiable tasks

## *Likelihood Self-Confidence*

Let's consider the sentence "very cool". The vocabulary is {cool, lol, ver, wow}. The LLM produces a probability distribution at each step.

In the case of likelihood self-confidence, we consider the individual probabilities of the generated tokens:

$$\begin{matrix} & \left\{ \begin{array}{l} p(\text{cool}) = 0.15 \\ p(\text{very}) = 0.60 \\ p(\text{lol}) = 0.10 \\ p(\text{wow}) = 0.15 \end{array} \right. & \text{very} & \left\{ \begin{array}{l} p(\text{cool}) = 0.75 \\ p(\text{very}) = 0.05 \\ p(\text{lol}) = 0.10 \\ p(\text{wow}) = 0.10 \end{array} \right. & \text{cool} & \left\{ \begin{array}{l} p(\text{cool}) = 0.10 \\ p(\text{very}) = 0.20 \\ p(\text{lol}) = 0.35 \\ p(\text{wow}) = 0.35 \end{array} \right. \\ \langle \text{b} \rangle & & & & & \end{matrix}$$

The probability of the sentence is then:  $p(\text{very cool}) = p(\text{very}) \times p(\text{cool}) = 0.60 \times 0.75$

For numerical stability, it is always better to take the sum of the log-probs.

# Self-Confidence for non-verifiable tasks

## *Entropy Self-Confidence*

In Entropy we consider the whole probability distribution at each step:

$$\begin{array}{l} \text{<b>} \left\{ \begin{array}{l} p(\text{cool}) = 0.15 \\ p(\text{very}) = 0.60 \\ p(\text{lol}) = 0.10 \\ p(\text{wow}) = 0.15 \end{array} \right. \quad \text{very} \left\{ \begin{array}{l} p(\text{cool}) = 0.75 \\ p(\text{very}) = 0.05 \\ p(\text{lol}) = 0.10 \\ p(\text{wow}) = 0.10 \end{array} \right. \quad \text{cool} \left\{ \begin{array}{l} p(\text{cool}) = 0.10 \\ p(\text{very}) = 0.20 \\ p(\text{lol}) = 0.35 \\ p(\text{wow}) = 0.35 \end{array} \right. \end{array}$$

The entropy  $H$  for a single probability distribution  $X$ :  $H(X) = - \sum_{p \in X} p \times \log p$

The entropy at the first step:  $H(\text{<b>}) = -(0.15 \times \log(0.15) + 0.60 \times \log(0.60) + 0.10 \times \log(0.10) + 0.15 \times \log(0.15))$

The average entropy:

$$H(\text{<b> very cool}) = \frac{1}{3} (H(\text{<b>}) + H(\text{very}) + H(\text{cool}))$$

# LLM as a Judge

LLMs are often better at judging than generating the right answers.

## **LLMs for scoring (Scalar Reward)**

You can ask LLMs to score answers

## **LLMs for verbal feedback (Generative Reward)**

Scalar rewards don't provide enough signals and nuances compared to verbal feedback.

# Reward Model

If you have a dataset of prompts  $\mathbf{x}$ , and the preferred (good) answers  $\mathbf{y}_p$  and rejected (bad) answers  $\mathbf{y}_r$ ,

Then you can train a Reward Model (RM) to assign high scores to good answers and low scores to bad answers:  $\text{score}(\mathbf{x}, \mathbf{y}_p) > \text{score}(\mathbf{x}, \mathbf{y}_r)$

There are many ways to train reward models. The most popular:  
**Bradley-Terry Model**

# Reward Model (Bradley-Terry)

- $\text{score}(\mathbf{x}, \mathbf{y}_p)$ : the score of the preferred answer given the prompt  $\mathbf{x}$
- $\text{score}(\mathbf{x}, \mathbf{y}_r)$ : the score of the rejected answer given the prompt  $\mathbf{x}$

Then, the probability of the preferred answer is:

$$p(y_p > y_r | x, \theta) = \frac{\text{score}(x, y_p)}{\text{score}(x, y_p) + \text{score}(x, y_r)}$$

Since it's a probability, we need to have positive scores.

$$\mathbf{c}_p = \text{Transformer}(\mathbf{x} \mathbf{y}_p)$$

$$\mathbf{c}_r = \text{Transformer}(\mathbf{x} \mathbf{y}_r)$$

$$\text{score}(\mathbf{x}, \mathbf{y}_p) = \exp(\mathbf{c}_p \cdot \mathbf{w}^T), \text{ where } \mathbf{w} \text{ is learnable vector}$$

$$\text{score}(\mathbf{x}, \mathbf{y}_r) = \exp(\mathbf{c}_r \cdot \mathbf{w}^T), \text{ where } \mathbf{w} \text{ is learnable vector}$$



# Reward Model (Bradley-Terry)

$$p(y_p > y_r | x, \theta) = \frac{\text{score}(x, y_p)}{\text{score}(x, y_p) + \text{score}(x, y_r)}$$

Since it's a probability, we need to have positive scores:

$$\mathbf{c}_p = \text{Transformer}(\mathbf{x}, \mathbf{y}_p)$$

$$\mathbf{c}_r = \text{Transformer}(\mathbf{x}, \mathbf{y}_r)$$

$$\text{score}(\mathbf{x}, \mathbf{y}_p) = \exp(\mathbf{c}_p \cdot \mathbf{w}^T)$$

$$\text{score}(\mathbf{x}, \mathbf{y}_r) = \exp(\mathbf{c}_r \cdot \mathbf{w}^T)$$

$$RM_\theta = \arg \max_{\theta} p(y_p > y_r | x, \theta)$$

**This is (again) Maximum Likelihood Estimation (MLE).**

The reward model is trained to maximize that the preferred answer is preferred over the rejected answer.

## Reward Model (Bradley-Terry)

$$p(y_p > y_r | x, \theta) = \frac{\text{score}(x, y_p)}{\text{score}(x, y_p) + \text{score}(x, y_r)}$$

**Exercise: derive the Bradley-Terry Model to get a Logistic Model:**

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

# Reward Model (Bradley-Terry)

$$p(y_p > y_r | x, \theta) = \frac{\text{score}(x, y_p)}{\text{score}(x, y_p) + \text{score}(x, y_r)}$$

Let call  $l_p, l_r$  the logits of the preferred answer and rejected answer respectively:

$$p(y_p > y_r | x, \theta) = \frac{\exp(l_p)}{\exp(l_p) + \exp(l_r)}$$

Using the rule  $A/B = (A/C)/(B/C)$ , we divide  $\exp(l_p)$  in the numerator and denominator:

$$p(y_p > y_r | x, \theta) = \frac{\frac{\exp(l_p)}{\exp(l_p)}}{\frac{\exp(l_p)}{\exp(l_p)} + \frac{\exp(l_r)}{\exp(l_p)}} = \frac{1}{1 + \frac{\exp(l_r)}{\exp(l_p)}}$$

# Reward Model (Bradley-Terry)

Using the rule  $A/B = (A/C)/(B/C)$ , we divide  $\exp(l_p)$  in the numerator and denominator:

$$p(y_p > y_r | x, \theta) = \frac{\frac{\exp(l_p)}{\exp(l_p)}}{\frac{\exp(l_p)}{\exp(l_p)} + \frac{\exp(l_r)}{\exp(l_p)}} = \frac{1}{1 + \frac{\exp(l_r)}{\exp(l_p)}}$$

Using the rule  $\exp(a) / \exp(b) = \exp(a - b)$ :

$$p(y_p > y_r | x, \theta) = \frac{1}{1 + \exp(l_r - l_p)}$$

$$p(y_p > y_r | x, \theta) = \frac{1}{1 + \exp(-(l_p - l_r))} = \sigma(l_p - l_r)$$

# Summary

## Scaling repeated sampling

- Scaling the ***number of attempts improves the performance*** of LLMs
- The performance as a function of **number of attempts  $k$**  is **determined by a power function**:  $c = \exp(ak^b)$
- **Large models improve more quickly** when scaling the number of attempts

## Reward Modeling

- For **verifiable tasks** (math, code, instructions), the reward is based on the **correctness** of the output
- For **non-verifiable tasks**:
  - **Self-confidence** can be a good signal (see lab on translation)
  - **LLM as a Judge** can be a good rewarder but they can be biased (prefer long answers)
  - One common solution is to **train a reward model on a preference dataset**
  - **Bradley-Terry** is a popular approach for training reward models. It is equivalent to applying **sigmoid** on the difference of scores between the preferred and rejected answer