AI Model Evaluation and benchmarking

- Accuracy (precision, recall)
- Performance (latency, throughput)
- NLP tasks: BLEU, ROUGE, perplexity.
- MLPerf
- Hugging Face Evaluate
- Understand how to evaluate LLM datasets such as Math500, GSM8k, and MMLU.
- Inferences per second / resource uttilization
- DeepEval (https://github.com/confident-ai/deepeval) Groq uses mlagility (https://github.com/groq/mlagility). OpenAI Evals (https://github.com/openai/evals). Simpleevals (https://github.com/openai/simple-evals)
- InspectEvals (https://github.com/AarushSah/inspect_evals)
- https://www.confident-ai.com/blog/llm-evaluation-metrics-everything-you-need-for-llm-evaluation
- https://learn.microsoft.com/en-us/ai/playbook/technology-guidance/generative-ai/working-with-llms/evaluation/list-of-eval-metrics

Model Optimization Techniques

- Quantization
- Pruning
- Distillation improve model efficiency

MLPerf

Two main tracks:

- Training - how fast a system can train a model from scratch
- Inferencing - how efficiently a system can run a pre-trained model on new data

Llama 2 70B: An MLPerf Inference Benchmark for Large Language Models

https://mlcommons.org/2024/03/mlperf-llama2-70b/#:~:text=For%20the%20MLPerf%20Inference%20v4,BLOOMZ%2C%20and%20Llama%202%2070B

Task Q&A

- Task force selected a question-answering scenario.

- Q&A is among the most common ways LLMs are used in actual applications (chatbots, tech support, knowledge bases).
- It is easier to evaluate and measure than, for example, a full interactive chatbot session with multiple turns.
- Each Q&A example is treated as an independent “prompt -> model output” pair, making measuring performance and accuracy more straightforward.

Accuracy is tricky for LLM

Unlike classifying an image as “dog” or “cat,” text generation can vary. Two answers might look different in wording but still be correct. Conversely, generation can go off-topic or produce incomplete sentences. Therefore, a robust metric is needed.

Rouge Scores

- a set of metrics designed to measure similarity between a generated text and a reference text. MLPerf uses **ROUGE-1**, **ROUGE-2**, and **ROUGE-L**.
  - Reference text: “The Eiffel Tower is located in Paris and is one of the most visited landmarks in the world.”
  - Generated text: “Located in Paris, the Eiffel Tower is a famous landmark visited by many people.”

ROUGE 1 (Uniform overlap)

- Reference:
  ["the", "eiffel", "tower", "is", "located", "in", "paris", "and", "is", "one", "of", "the", "most", "visited", "landmarks", "in", "the", "world"]
- Generated:
  ["located", "in", "paris", "the", "eiffel", "tower", "is", "a", "famous", "landmark", "visited", "by", "many", "people"]
- Overlap:
  ["located", "in", "paris", "the", "eiffel", "tower", "is", "visited"] → 8 overlapping unigrams
  ROUGE-1 Recall = 8 / 18 ≈ 0.44

ROUGE 2 (Bigram Overlap) - compare consecutive word pairs (bigrams)
• Reference Bigrams:
["the eiffel", "eiffel tower", "tower is", "is located", "located in", "in paris", "paris and", "and is", ..., "the world"]

• Generated Bigrams:
["located in", "in paris", "paris the", "the eiffel", "eiffel tower", ..., "many people"]

• Overlap:
["located in", "in paris", "the eiffel", "eiffel tower"] → 4 overlapping bigrams
• ROUGE-2 Recall = 4 / (Total reference bigrams)
(Recall measures how much of the reference was captured)

ROUGE L (Longest Common Subsequence) - look for the longest sequence of words that appear in both texts in the same order, not necessarily
• LCS:
["located", "in", "paris", "the", "eiffel", "tower", "is", "visited"]

• Length of LCS: 8
• ROUGE-L Recall = 8 / 18 ≈ 0.44

Quantization of Models

- We would have the model Llama 2 70B as the gold model "reference version" 32-bit-floating-point. Why is this the standard? -> Because
  - Let's say a neural network weight is 0.123456789. With FP32, it can store that number with very high precision. With lower-precision formats (like FP16 or INT8), it may become something like 0.1234 or even 0.12, losing detail.
  - Why use anything other than FP32? -> Slow and takes more memory
  - That's why MLPerf checks whether these smaller, faster models still stay close to the FP32 "gold standard" using metrics like ROUGE.
  - FP16 (16-bit) or INT8 (8-bit integers) are used to speed up inference and save memory.
    - INT8 quantization: Take float values -> scale and round them into 8-bit-integers (**quantization**). During inference, we dequantize. Convert those int8 values back to float approximation
- It runs with the highest numerical precision, so it's assumed to produce the most accurate, high-quality output.
- When someone submits a faster or smaller version of the model (e.g., using int8 quantization), MLPerf wants to make sure it's not losing too much quality.

To check that a submitted model's output is close enough to the FP32 reference, MLPerf compares the ROUGE scores of the two outputs.

Category	Requirement	What it Means
Category 1	≥ 99.9% of reference ROUGE scores	Almost identical to gold standard

Category	Requirement	What it Means
Category 2	≥ 99.0% of reference ROUGE scores	Still very close, but a tiny bit more relaxed

Prompt	Reference (LLaMA 2 70B FP32)	Submission Model Output	ROUGE-L
1	“The quick brown fox…”	“The quick brown fox…”	100%
2	“It is raining today.”	“It’s raining today.”	97.2%
3	“Let’s explore the stars.”	“We explore stars.”	90.1%
…	…	…	…
Avg: 98.6%			

So in this case, your model scores 98.6% vs. the reference. If the threshold is 99.0%, it doesn’t pass.

Performance Measurement

Tokens per Second

Measuring how many tokens the model can generate (and process) per second is more insightful than queries per second for LLMs. Why?

- Each query can have different lengths of input (prompt) and output (answer).
- Focusing on *tokens/second* normalizes the measurement, regardless of how short or long the prompts and answers are.

Latency Constraints (Server Scenario)

In a “server” setting, queries arrive randomly (simulating real-world user traffic). MLPerf applies latency requirements to make sure each system can handle queries at a certain speed:

1. TTFT (Time to First Token): Must be ≤ 2 seconds.
   - This is how fast the model can start responding.
2. TPOT (Time per Output Token): Must be ≤ 200 ms.
   - How quickly can it generate each subsequent token?
   - 200 ms per token approximates a human reading rate of ~240 words per minute. This is a rough baseline to ensure responsiveness.

Preventing Cheats

First Token Consistency

What it means:
The first token your model outputs must be exactly the one that is timed and reported.

What cheat it prevents:
If someone precomputes or “cheats” by generating the first token ahead of time (before the official benchmark timer starts), it would make the model seem faster than it really is.
This rule prevents “head start” cheating.

EOS (End-of-Sequence) Token

What it means:
There should only be one EOS token, and it must come at the very end of the model’s output.

What cheat it prevents:
A model might try to insert multiple EOS tokens early to trick the benchmark into stopping the generation prematurely — this would lower latency and make the system look faster.
This rule ensures the generation runs its full course.

Token count verification

Closed vs Open division

MLPerf Inference has **two divisions** for submissions:

- **Closed**: Everyone must use the same reference model and dataset. Purpose: isolate hardware/software performance without model optimization trickery.
- **Open**: Allows tweaking or optimizing the model architecture or weights. Purpose: highlight gains from both software and model innovations.

LoadGen

MLPerf uses a standard benchmarking harness called **LoadGen**, which:

- Sends input data to the system under test (SUT).
- Measures latency, throughput, and accuracy.
- Supports multiple test modes like:
  - **Single-stream** (one query at a time),
  - **Multi-stream** (parallel queries),
    - Measures how many streams the system can maintain within a latency threshold.
  - **Server mode** (queries arrive randomly), Poisson distribution
    - **Latency percentile** (e.g., 99% of responses must be under X ms).
    - **Throughput under latency constraint**.
  - **Offline** (throughput-focused, like batch processing).

In LLM inference, **server mode** is the most realistic for chatbot-style deployments, as it mimics user traffic.

Inspect AI

A variety of evaluation types:

- Standard benchmarks
- Agent evaluations (planning, memory, tool usage, simulating real-world agent behavior)
- Multimodal evaluations (assess LLMs on tasks involving images, audio, and video)
- Human baselining

Evaluation process:

- Task definition
  - Each task contains
    - dataset (set of inputs question, prints, problems) and expected outputs. GAIA, SWE-Bench
    - Solver , model generating a response.
      - Simple prompting
      - Advanced techniques like chain-of-thought prompting, where the model is encouraged to reason step-by-step.
      - Custom workflows, such as multi-turn dialogues or tool-use, depending on the evaluation's needs.
    - **Scorer**: The mechanism for assessing the model's output against the expected result. Scorers can use:
      - Exact matching for tasks with definitive answers (e.g., multiple-choice questions).
      - Model-graded scoring, where another LLM evaluates the response quality (e.g., for open-ended questions).
      - Custom metrics tailored to specific tasks, like functional correctness for code generation.
- Running eval
- Prompt process
  - The solver component handles how prompts are sent to the model. By default, it processes the dataset's inputs (prompts) and collects responses, which are then passed to the scorer for evaluation. The efficiency of this step depends on both the solver's implementation and the model provider's capabilities.
- Scoring and output

Inference Metrics

When inference, MLPerf uses two main types of tests:

Performance (throughput)

- Metric: Queries per second / tokens per second / inferences per second
  - Measure how many predictions a system can make per unit time
  - Queries per second: If model can handle 1000 QPS -> means can serve 1000 people sending questions every second
  - Samples per seconds: Instead of queries, how many data samples (image, audio files) are processed per second
- Per user or system wide
  - Per user: Tokens generated per second for a single stream
  - System wide: Total tokens or queries per second a server can handle across all users
- Higher throughput affects **deployment cost**
  - Higher throughput: More work done per unit time. If model can handle more tokens/sec, can serve more users or requests using the same hardware
  - Fewer servers needed to meet demand
  - Shorter runtime per job -> less time billed

Model Setup	Tokens/sec	Total Time	Cost (on-demand LPU})
Slower model	100	1000 sec	$0.83 (e.g., 1 hour LPU = $3)
Faster model	500	200 sec	$0.17 (⅓ hour LPU)

Fewer concurrent servers = Less infra cost
Let's say:

- Each model server can handle 5 inferences/sec.
- You need to serve 500 QPS during peak hours.
  Need 500 / 5 = 100 servers
  if you optimize throughput to 25 inferences/sec:
  500 / 25 = 20 servers
- Server and offline scenarios:
  - Server scenario - mimics real-world unpredictable request loads
    - chatbot, search enginge, voice assistant
    - requests come in random - different users sending queries at differnet times

        - goal: can model keep up with live user traffic? You're running an AI assistant for a website. If 500 users ask questions at the same time, can your model respond to all 500 within a second?
    - Offline scenario - batch process a large dataset at once
        - processing large amount of data at once, not live
        - goal: how fast can your system process a bulk of inputs? You want to label 10 million photos using a trained model. Throughput here is: how many photos can be labeled per second?

Latency (Response time)

- Metric: latency in milliseconds
    - Measures how quickly the system returns a result after a request. Moment it receives input and moment it returns results
- Important for real-time applications like chatbots
- Lower latency = faster response = better user experience (especially in real-time apps).

Measures:

- Time to First Token (TTFT)
    - Delay from when a user sends a prompt to when the model produces the first word/token of its reponse
- Time per token (TPOT)
- TPOT (infer-token latency) - measures how fast the model continues generating tokens after the first one
    - 100 ms/token means the model generates about 10 tokens per second for each user - which translates to 450 words per minute

Overall response time: TTFT + TPOT × number of tokens

Different use cases have different latency requirements:

- Chatbots aim for low TTFT to feel responsive
- Batch processing (offline) - might tolerate higher latency but care about throughput

BLEU vs ROUGE

ROUGE

- **ROUGE** evaluates how much of the **reference text** is captured in the **generated text**.
- It’s based on **recall** -checks how much of the "important stuff" from the ground-truth/reference is recovered by the model output.

- Good for LLM because it generates fluent but varied responses

Pros:

- Good at judging coverage. Better for abstractive tasks (summarization, paraphrasing)
  Cons:
- May reward verbose answers if they happen to include more reference terms
- Doesn't always penalize irrelevant extra informatino

BLEU

- **BLEU** evaluates how much of the **generated text** matches the **reference text**. Using **n-gram overlap** (i.e., matching word sequences)
- It's based on **precision**, meaning it checks how much of the output is correct based on the reference.

Example:

- A reference sentence (what a human would say)
- A candidate sentence (what the machine translation or model output is)

```
reference = ['the', 'cat', 'is', 'on', 'the', 'mat']
candidate = ['the', 'cat', 'is', 'on', 'mat']
```

Extract n-grams (typically up to 4-grams)

Unigrams (1-grams):

- Ref: `['the', 'cat', 'is', 'on', 'the', 'mat']`
- Cand: `['the', 'cat', 'is', 'on', 'mat']`

Overlapping: `'the', 'cat', 'is', 'on', 'mat'` → 5 matches

Bigrams (2-grams):

- Ref: `['the cat', 'cat is', 'is on', 'on the', 'the mat']`
- Cand: `['the cat', 'cat is', 'is on', 'on mat']`

Matches: `'the cat', 'cat is', 'is on'` → 3 matches out of 4

```
precision = (# of overlapping n-grams) / (# of candidate n-grams)
```

Brevity Penalty

```
BP = 1 if candidate_len > reference_len
BP = exp(1 - reference_len / candidate_len) if candidate_len <=
reference_len
```

In our case:

- candidate: 5 words
- reference: 6 words

```
BP = exp(1 - 6/5) = exp(-0.2) ≈ 0.8187
```

Final BLEU:

```
BLEU = BP × exp(∑ weights × log(precision_n))
```

How to evaluate LLM datasets such as Math500, GSM8k, MMLU

Dataset characteriscs

- GSM8k: 8.5K grade school math world problems. Each problem require multi step reasoning. Eval here focuses on whether final numerical answer exactly matches the correct solution
- Math500 - competition level math problems. Eval involve exact match of final answer but also scrutiny of reasoning process
- MMLU - cover 57 academic subjects (stem, humanities, social sciences). Performance = percentage of correctly answered questions (evaluated under zero shot or few shot)

Evaluation Methodologies

Metric Selection

- *Exact Matching:* For datasets like GSM8K and Math500, the most straightforward metric is exact match accuracy, where the model's final numeric answer must be identical to the ground-truth answer.
- *Multiple-Choice Accuracy:* For MMLU, the standard metric is accuracy—the proportion of questions answered correctly.
- *Chain-of-Thought and Intermediate Steps:* In math reasoning tasks, especially on datasets like GSM8K, the quality of the intermediate reasoning steps can be crucial. Researchers sometimes evaluate these steps to better understand whether a model is truly reasoning or merely guessing the final answer.

Verification and Robustness Techniques

- *Verifier Models:* Some approaches involve training separate verifier models to assess candidate solutions, which can improve the overall performance y selecting the most plausible answer from multiple generated responses.
- *Best-of-N Sampling:* To account for the stochastic nature of language models, many evaluations (especially in math tasks) use “best-of-N” sampling, where multiple responses are generated and the best one (according to a verifier or via exact matching) is chosen.
- *Error Analysis:* Beyond simple accuracy, a deep evaluation might analyze the types of errors (e.g., arithmetic miscalculations vs. logical missteps) to inform model improvements.

Python and GoLang's role in eval pipelines

When Python Concurrency Helps LLM Evaluation

- **I/O-bound tasks**:
  - Fetching data from remote sources (e.g., downloading, API requests)
  - Pushing or pulling from databases or object storage.
  - Asynchronous calls to an LLM API.
- **Batch Inference**:
  - Spawning multiple processes to load parts of the data and query the model or API. This is especially useful if you have a local GPU server that can handle multiple processes or if you’re calling an external inference service in parallel.
- **Parallel Metrics Computation**:

  - If you have computationally heavy metrics (e.g., big textual comparisons, large embeddings for similarity checks), using multiprocessing can help utilize multiple cores.

  **When Go Concurrency Helps LLM Evaluation*&*
- **High-volume concurrency**: If you need to evaluate thousands (or more) of prompts in parallel, Go can handle these tasks very efficiently through goroutines
- **Pipelines with Channels**:
  - You can stream tasks (e.g., prompts) from a source channel through multiple worker goroutines (model inference, metrics calculations) and then pass results to another stage (logging, analysis).
  - This makes it straightforward to design a multi-stage pipeline that looks like an assembly line.

Example Go Pipeline

Imagine you have three stages:

1. **Prompt Generation** (stage 1): Reads from a dataset or file and generates prompts to be tested.
2. **Inference** (stage 2): Sends prompts to the model or an API, gathers responses.
3. **Evaluation** (stage 3): Compares generated responses to references, calculates metrics, and logs results.

```
func promptGenerator(prompts []string, out chan<- string) {
    defer close(out)
    for _, prompt := range prompts {
        out <- prompt
    }
}

func inferenceWorker(in <-chan string, out chan<- InferenceResult) {
    defer close(out)
    for prompt := range in {
        // call model or API
        response := callModel(prompt)
        out <- InferenceResult{Prompt: prompt, Response: response}
    }
}

func evaluate(in <-chan InferenceResult, done chan<- bool) {
    for result := range in {
        // compare result.Response to a reference
        // store or print metrics
    }
    done <- true
}

func main() {
    promptChan := make(chan string)
    inferenceChan := make(chan InferenceResult)
    done := make(chan bool)

    prompts := []string{"Question 1", "Question 2", ...}

    // Stage 1
    go promptGenerator(prompts, promptChan)

    // Stage 2
    go inferenceWorker(promptChan, inferenceChan)

    // Stage 3
    go evaluate(inferenceChan, done)

    <-done  // Wait for evaluation stage to finish
}
```