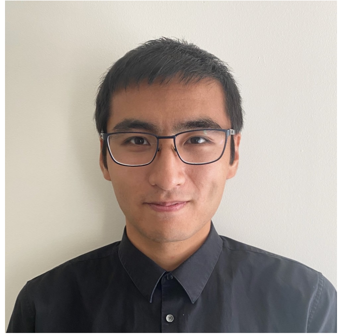
On the Query Complexity of Verifier-Assisted Language Generation





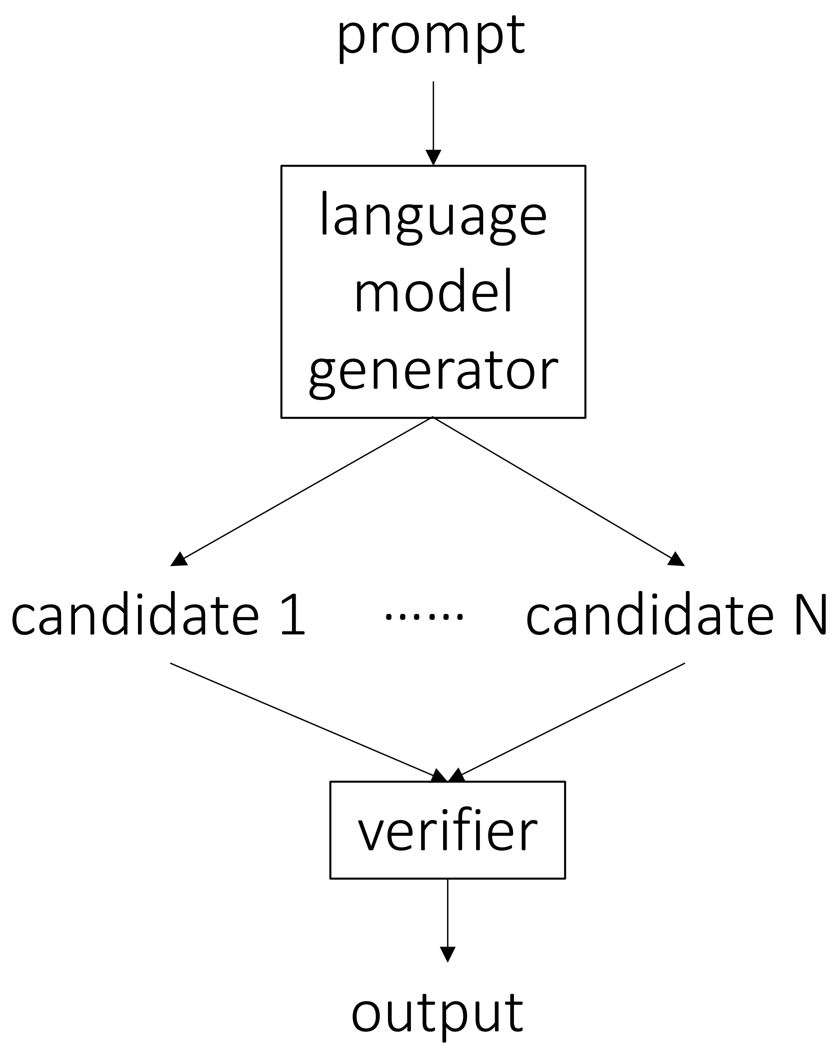


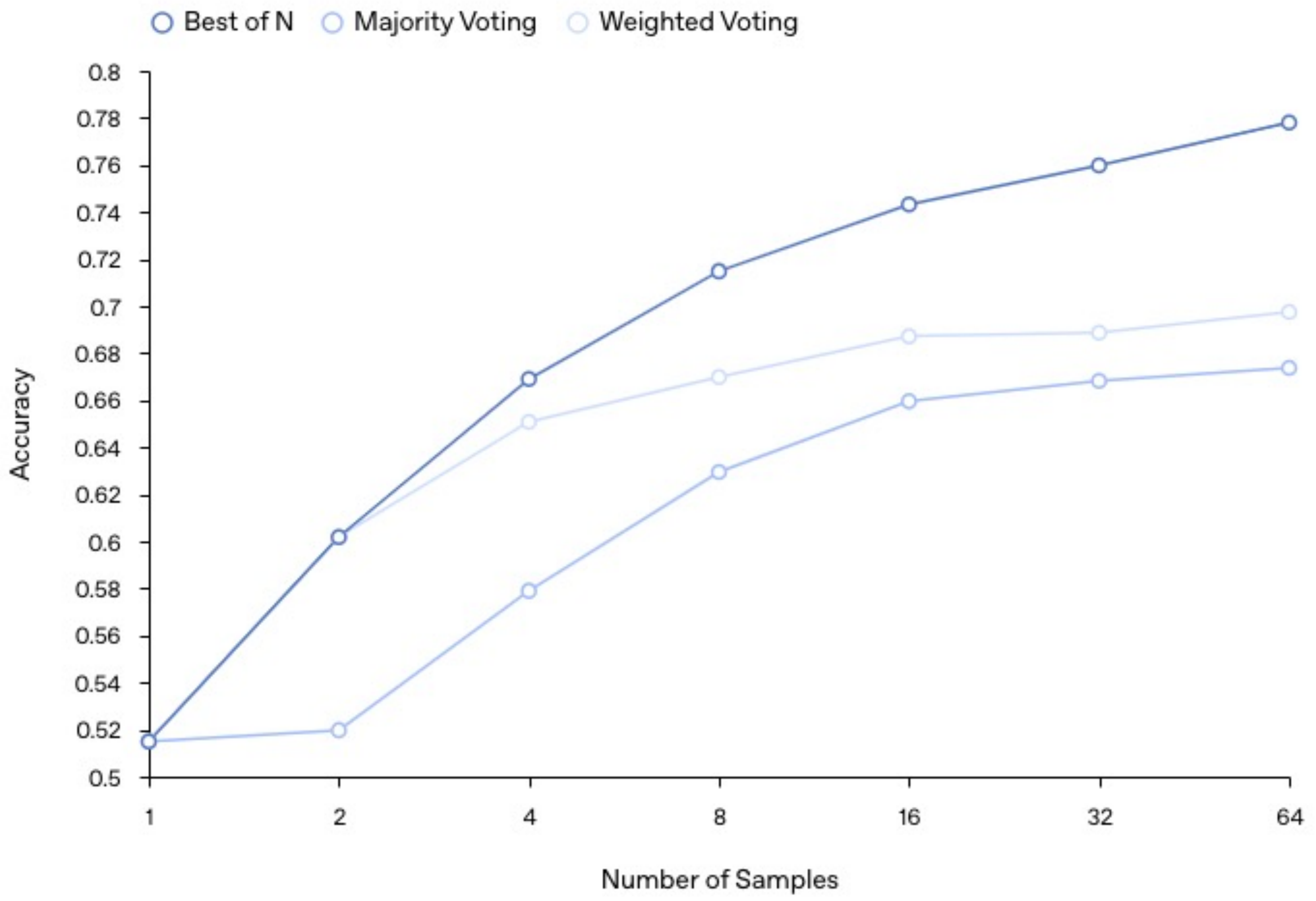

Edoardo Botta[1] Yuchen Li[1] Aashay Mehta[1] Jordan T. Ash[2] Cyril Zhang[2] Andrej Risteski[1]

([1]Carnegie Mellon University [2]Microsoft Research NYC)

https://arxiv.org/abs/2502.12123 (ICML 2025)

Background: verifier-assisted language generation





Karl Cobbe et al, 2021, Training Verifiers to Solve Math Word Problems

Figure on the right: Jason Wei et al, 2025, BrowseComp

Overview of our results

Key takeaway: constrained language generation without a verifier is provably hard, but **verifiers** (which check partial outputs) can help.

Constrained generation task

Information-theoretic lower bound

Computational lower bound

Theoretical framework

valid [ ] ( ) [( )]

([{ }])

invalid [ ) ( ] [( ])

([ ]

codellama

Empirical study

Setup: constrained generation task

Autoregressive generator oracle $\mathcal{O}$: given string $s \in \Sigma^*$, predict and sample from next-token distribution $\mathcal{O}(s): \Sigma \to \mathbb{R}_+$.

Constrained generation task $(\Sigma, A, \mathcal{O})$: find $s \in A$ s.t. $P_{\mathcal{O}}(s) > 0$. If no such s exists, return FAIL.

Oracle complexity: expected number of calls to $\mathcal{O}$ to solve the constrained generation task

- Σ: vocabulary
- $\mathcal{O}$: autoregressive generator oracle
- $P_{\mathcal{O}}: \Sigma^* \to \mathbb{R}_+$: distribution over strings predicted by $\mathcal{O}$
- $A \subset \Sigma^*$: constraint set

Constrained generation is hard without a verifier

Theorem 1 (information-theoretic lower bound, informal):
There exists a constrained generation task for which any (possibly randomized) algorithm has (expected) oracle complexity at least exponential in seq length.

Theorem 2 (computational lower bound , informal):
There exists a constrained generation task which is NP-hard.

Incorporating verifiers into constrained generation

Verifier $V: \Sigma^* \to \{0,1\}$: $\forall s \in \Sigma^*$, $V(s) = 1$ if and only if $\exists s' \in \Sigma^*$ s.t. $s \circ s' \in A$.

Rejection sampling: repeatedly generating complete strings s according to $P_{\mathcal{O}}$, until $V(s) = 1$.

Tokenwise rejection sampling: given prefix s, sample next token $t \sim \mathcal{O}(s)$, until $V(s \circ t) = 1$, then proceed to next token.

- **Backtrack**: additionally, whenever $V(s \circ t) = 0$, resample the last B positions of s (allowed $\leq Q$ times)

- Σ: vocabulary
- $\mathcal{O}$: autoregressive generator oracle
- $P_{\mathcal{O}}: \Sigma^* \to \mathbb{R}_+$: distribution over strings predicted by $\mathcal{O}$
- $A \subset \Sigma^*$: constraint set
- V: verifier
- $\circ$: concatenation of strings
- Q: backtrack quota
- B: stride

Tokenwise rejection sampling is efficient

Proposition 1 (informal): there exists a constrained generation task s.t.

- The expected oracle complexity of rejection sampling is **exponential** in seq length, and
- The expected oracle complexity of **tokenwise** rejection sampling is **linear** in seq length.

- Σ: vocabulary
- $\mathcal{O}$: autoregressive generator oracle
- $P_{\mathcal{O}}: \Sigma^* \to \mathbb{R}_+$: distribution over strings predicted by $\mathcal{O}$
- $A \subset \Sigma^*$: constraint set
- V: verifier
- $\circ$: concatenation of strings

Experiment 1: generating strings in Dyck grammar

- **Constrained generation task**: given a prefix, generate a completion to form a valid Dyck grammar string
- **Dyck grammar constraints** A: language of balanced parentheses
- **Generator** $\mathcal{O}$: pre-train an autoregressive Transformer from scratch on Dyck grammar strings
- **Verifier** V: one-layer MLP trained from scratch for binary classification
  - Feature = generator representation of a prefix
  - Label = 1 if the prefix is grammatical, 0 otherwise

valid [ ] ( ) [( )]

([{ }])

invalid [ ) ( ] [( ])

([ ]

Experiment 1: generating strings in Dyck grammar

top_p	quota	stride	#errors ± std err
0.9	0	0	240.0 ± 5.177
	4	4	179.4 ± 1.020
1.0	0	0	461.8 ± 8.304
	4	4	200.0 ± 3.225

- Q: backtrack quota
- B: stride
- **top_p**: controls nucleus sampling truncation

Tokenwise rejection sampling with backtracking reduces completion errors

Experiment 2: generating Python test cases

- **Constrained generation task**: given the codes for a simple Python function (i.e. list append), generate test cases (assert statements)
  - **Eval metric**: distinct accuracy $Acc_{distinct}$: the number of distinct correct test cases generated, divided by the total number requested
- **Generator $\mathcal{O}$**: pre-trained CodeLlama[1]
- **Verifier V**: one-layer MLP for binary classification
  - Feature = generator representation of a prefix
  - Label = 1 if the prefix is grammatical, 0 otherwise

1. Baptiste Rozière et al, 2023, Code Llama: Open Foundation Models for Code

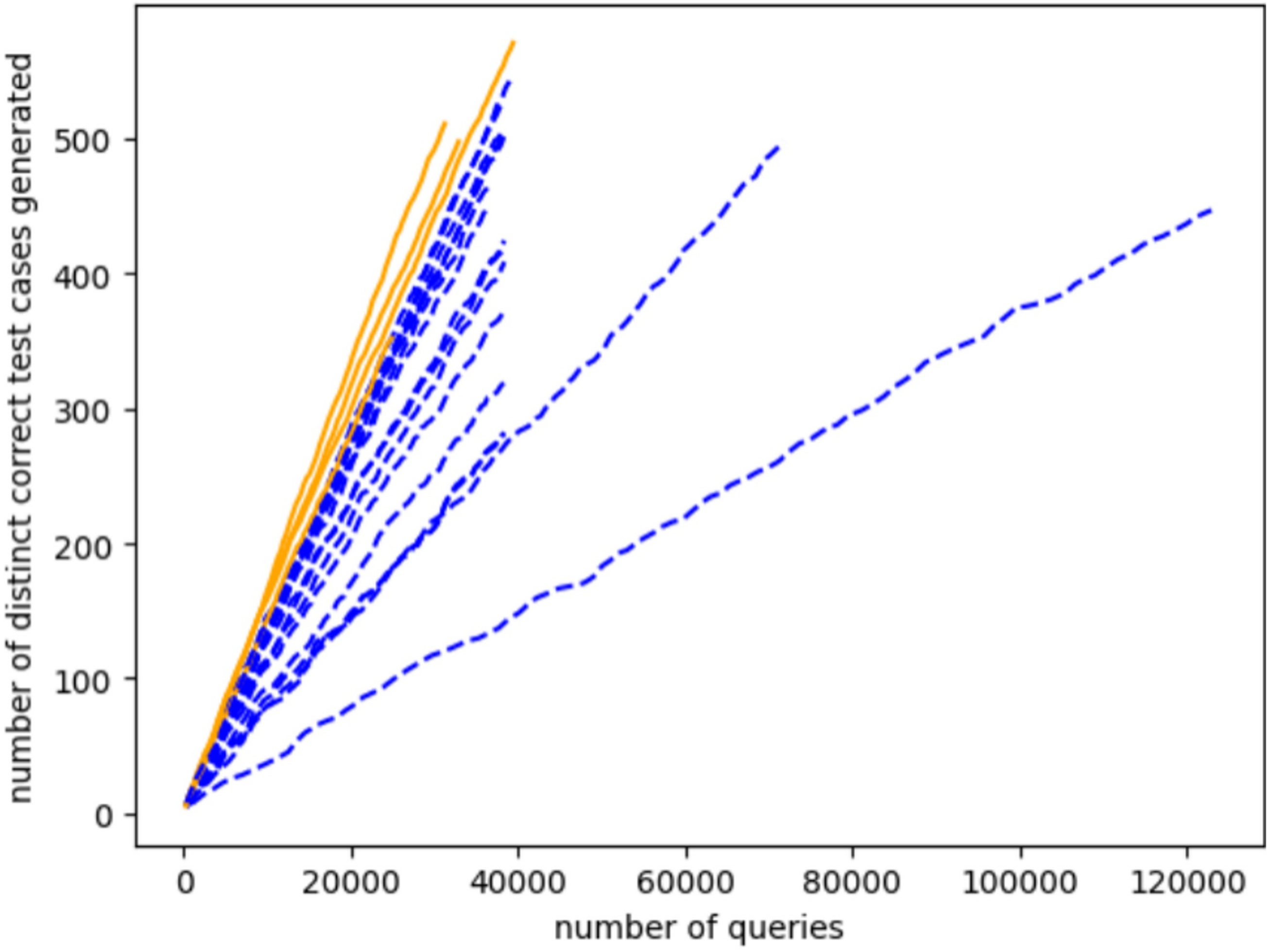
Experiment 2: generating Python test cases

quota	stride	top_p	T	block BoN	$\text{Acc}_{\text{distinct}} \pm$ std err
4	4	0.95	1.0		0.714 ± 0.011
0		0.95	1.0	2	0.684 ± 0.038
0		0.95	1.0		0.660 ± 0.042
0		0.95	1.0	4	0.623 ± 0.036
0		0.95	1.0	8	0.559 ± 0.038
4	4	1.0	1.0		0.639 ± 0.061
4	10	1.0	1.0		0.622 ± 0.046
0		1.0	1.0		0.504 ± 0.025
4	4	1.0	1.2		0.440 ± 0.026
0		1.0	1.2		0.269 ± 0.025
0		0.0	1.0		0.013 ± 0.000

- Q: backtrack quota
- B: stride
- **top_p**: controls nucleus sampling truncation
- T: sampling temperature
- **block BoN**: block best-of-N (baseline)

Tokenwise rejection sampling with backtracking improves distinct accuracy

Experiment 2: generating Python test cases



- Orange: Tokenwise rejection sampling with backtrack
- Blue: baselines, including various nucleus sampling top_p, sampling temperature T, and block best-of-N

Tokenwise rejection sampling with backtracking improves query efficiency

Summary

Contact: yuchenl4@cs.cmu.edu
https://arxiv.org/abs/2502.12123 (ICML 2025)

Key takeaway: constrained language generation without a verifier is provably hard, but **verifiers** (which check partial outputs) can help

Theory: query complexity, information-theoretic and computational lower bound

Experiments: Tokenwise rejection sampling with backtracking improves accuracy, diversity, and query efficiency for generating Dyck grammar and Python test cases

Open questions for future work:
1. Theory when verifiers are imperfect
2. Maintaining calibration of the distribution of elements in the constrained support