VALIDATING LARGE LANGUAGE MODELS WITH RELM

Michael Kuchnik [1] **Virginia Smith** [1] **George Amvrosiadis** [1]

ABSTRACT

Although large language models (LLMs) have been touted for their ability to generate natural-sounding text, there are growing concerns around possible negative effects of LLMs such as data memorization, bias, and inappropriate language. Unfortunately, the complexity and generation capacities of LLMs make validating (and correcting) such concerns difficult. In this work, we introduce `ReLM`, a system for validating and querying LLMs using standard regular expressions. `ReLM` formalizes and enables a broad range of language model evaluations, reducing complex evaluation rules to simple regular expression queries. Our results exploring queries surrounding memorization, gender bias, toxicity, and language understanding show that `ReLM` achieves up to $15\times$ higher system efficiency, $2.5\times$ data efficiency, and increased statistical and prompt-tuning coverage compared to state-of-the-art ad-hoc queries. `ReLM` offers a competitive and general baseline for the increasingly important problem of LLM validation.

1 INTRODUCTION

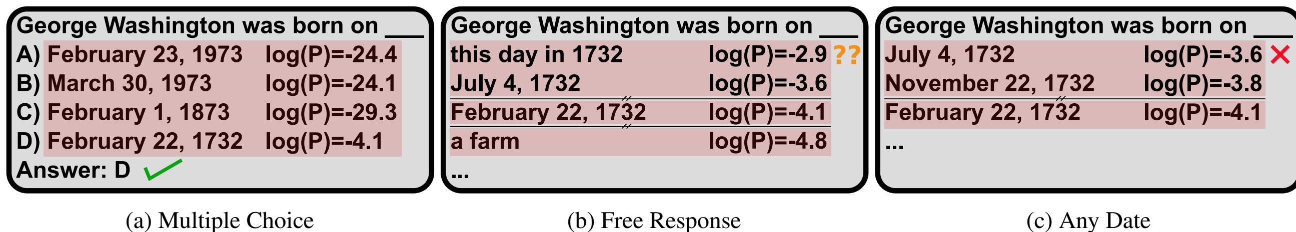
Large language models (LLMs), such as GPT-3 (Brown et al., 2020) and PaLM (Chowdhery et al., 2022), are a popular tool for many natural language processing tasks. While it is well understood that these models are expensive to train and deploy, there are growing concerns around even the seemingly simple problem of *validating* LLM behavior (Bowman & Dahl, 2021; Srivastava et al., 2022). Validation is particularly important in that LLMs may have unintended effects (Bommasani et al., 2021), such as returning memorized training data (Carlini et al., 2023), encoding bias in results (Bender et al., 2021), and generating inappropriate content (Gehman et al., 2020; Ousidhoum et al., 2021; Brown et al., 2020). While there have been extensive efforts to perform such testing on LLMs, the tests are written in an ad-hoc manner, where test maintainers explicitly code out the test's logic using LLM-specific utilities (Kiela et al., 2021; Srivastava et al., 2022; Gao et al., 2021a). Test users then add tasks by extending the existing code or supplying template parameters for that code (e.g., via millions of lines of JSON). In these approaches, it is up to the user to convert the expected LLM behavior into a sequence of test vectors that can be executed, making it difficult to maintain, modify, and extend test functionality.

As an example, consider testing if an LLM knows the birth date of George Washington via a fill-in-the-blank query: `George Washington was born on` ______, as shown in Figure 1. Today's standard solution is to prompt the model with a "multiple-choice" assessment using a few dates (Srivastava et al., 2022) (Figure 1a). However, such "closed-choice" assessments have shortcomings: with 12 months in a year, 31 days in a month, and thousands of years to consider, there are millions of candidates, and thus, the selection of dates can bias the evaluation. For example, the selected answer will always change if a more probable candidate was introduced, making the test prone to false positives. The alternative, sampling open-domain "free-response" strings (Figure 1b), is more challenging to test, as it requires grading arbitrary strings relative to all representations of the correct date. While the former case may bias performance, the latter seems impossible to control: every possible response, including `this day in 1732` or `a farm`, must be considered and graded, which requires carefully tuning the evaluation to avoid false positives and false negatives (Prager, 2006; Roberts et al., 2020). The inability to control the response of the LLM is in itself a testing bottleneck, motivating structured queries (Figure 1c), which are *guaranteed* to be drawn from the full set of expected strings (e.g., those of the pattern **`<Month> <Day>, <Year>`**) without unexpected responses.

In this work, we introduce the first queryable test interface for LLMs. Our system, `ReLM`, is a Regular Expression engine for Language Models and enables executing commonly-occurring *patterns*—sets of strings—with *standard regular expressions*. `ReLM` is the first system expressing a query as the complete *set* of test patterns, empowering practitioners to directly measure LLM behavior over sets too large to enumerate. The key to `ReLM`'s success is its ability to compactly represent the solution space via a graph representation, which is derived from regular expressions, compiled

[1]Carnegie Mellon University. Correspondence to: Michael Kuchnik <mkuchnik@cmu.edu>.

Proceedings of the 6th MLSys Conference, Miami Beach, FL, USA, 2023. Copyright 2023 by the author(s).



(a) Multiple Choice (b) Free Response (c) Any Date

Figure 1: Testing an LLM's knowledge of George Washington's birth date (LLM predictions highlighted in pink). (1a) Using 4 out of all possible dates and ranking them. An LLM classifying on year alone is sufficient to guess the answer correctly, limiting test resolution. Note that the answers can alternatively be encoded in the prompt, with predictions over the answer's letter. (1b) Allowing the LLM to output any completion, resulting in unexpected responses. (1c) A structured query over all dates of the form `<Month> <Day>, <Year>`, obtaining the specificity of 1a with the generality of 1b. Our approach finds that, among a search space of all dates, GPT-2XL's highest ranked prediction is incorrect, though the correct prediction is in the top 10. The same approach shows that the small variant of GPT-2 cannot discern the date even in 1a.

to an LLM-specific representation, and finally executed. As a result, users do not have to understand implementation details of the LLM itself—tests have the same effect as if all string possibilities were materialized. In addition to introducing `ReLM`, we demonstrate how various LLM evaluation tasks can be mapped onto `ReLM`'s string patterns.

For `ReLM` users, programming a validation task consists of two components: 1) formally specifying a set of strings of interest via a regular expression and 2) instructing the engine on how to enumerate and score the strings. For example, memorization tests can be expressed as finding a sample of training data in the LLM, bias can be expressed as the co-occurrence of data in sampled LLM sentences, toxicity can be expressed as finding insults within LLM-generated sentences, and knowledge can be expressed by assigning higher likelihood to the correct answer. However, unlike enumerating the sequences in a pattern, `ReLM`'s queries are *succinctly* defined with a graph representation, allowing `ReLM` to scale to queries with billions of strings in a few lines of code. Using the examples of URL memorization, gender bias, toxicity, and language understanding tasks with GPT-2 models, we demonstrate that `ReLM` is both efficient and expressive in executing common queries—dramatically lowering the bar to validate LLMs. Our contributions are:

(1) A formalization of regular expressions on LLM predictions. Unlike multiple choice questions, which are few and enumerable, regular expressions can model sets of infinite size. Unlike free response questions, which may result in spurious answers, `ReLM`'s results are always well-defined.

(2) An identification and construction of two commonly used classes of LLM inference queries, *conditional* and *unconditional* generation. For unconditional generation, we find that a fixed query string can be represented by many token sequences, motivating a compressed representation. To the best of our knowledge, we are the first to support these alternative encodings via automata.

(3) The design and implementation of a regular expression to inference engine, which efficiently maps regular expressions to finite automata. We implement both shortest path and randomized graph traversals that yield outputs in seconds at competitive GPU utilizations.

(4) An evaluation of memorization, gender bias, toxicity, and language understanding tasks using GPT-2 models, where we demonstrate the utility of `ReLM` within the context of LLM validation. `ReLM` achieves a $15\times$ speedup or $2.5\times$ higher data efficiency over traditional approaches in memorization and toxicity finding, respectively. As a diagnostic tool, `ReLM` exposes testing over character- and token-level transformations, measuring the robustness of bias to input representations. As a tuning tool, `ReLM` effortlessly supports common prompt tuning transformations necessary for state-of-the-art zero-shot performance, enabling practitioners to quickly iterate on their prompt design.

2 Background and Related Work

This work, being a regular expression engine for LLMs, primarily spans classical formal language theory as well as modern LLM architectures. We motivate the problem of LLM validation in Section 2.1 and then discuss how prompts and tests are structured in Section 2.2. In Section 2.3, we re-cast the specification of LLM prompts and LLM outputs using formal languages. Finally, we focus on the semantics of testing for particular behavior in autoregressive models in Section 2.4.

2.1 The Shifting Landscape of LLM Validation

The Transformer architecture (Vaswani et al., 2017) resulted in the backbone of many LLMs. Autoregressive models, such as the GPT family (Radford et al., 2018; 2019; Brown et al., 2020), were able to adapt to downstream tasks by representing the task specification itself inside the input (i.e., in the prompt) (Rocktäschel et al., 2016). Masked

models, of the BERT (Devlin et al., 2019) family, instead filled the role of transfer via fine-tuning. Subsequent work unified the interface of transfer-focused models such that all inputs and outputs are strings (Raffel et al., 2020). Recently, LLMs have exceeded human capabilities on many benchmarks (Wang et al., 2019), raising concerns that standard benchmarks are no longer sufficient for tracking progress in the field (Bowman & Dahl, 2021; Srivastava et al., 2022). These trends point to a need for more rigorous and holistic validation efforts (Liang et al., 2022), where an LLM's performance is measured over tasks spanning: 1) strings in both input and output space, 2) enough difficult content to generate a "grading curve", and 3) a variety of model behavior such that performance is broken down by area (e.g., memorization, bias, toxicity, knowledge). There are currently primarily two ways to grade LLMs in a black-box fashion: *multiple choice* and *free response* questions (Srivastava et al., 2022), which we discuss formally below (§2.4). Multiple choice questions present typically 2–10 completions to the LLM, which are scored, and the most likely response is used as the LLM's answer. Free response questions allow the LLM to generate any completion. In both cases, the LLM's answer is checked (usually verbatim) against a reference solution to assign a score. LLM validation is thus analogous to software-engineering's unit-tests over strings in the LLM's output space (e.g., over 10^{4000} for GPT-2).

2.2 Specifying the Input/Output of LLMs

In LLMs, both the inputs and outputs of the model are often strings. The use of strings as a data representation makes it possible to form predictions over mostly arbitrary objects, by simply converting their representation into a string form. The act of forming a string input is called *prompt engineering* and is an active research area (Gao et al., 2021b; Liu et al., 2021; Schick & Schütze, 2021; Jiang et al., 2020; Reynolds & McDonell, 2021). The act of testing a string output has many names, but most forms of testing can be viewed as generalizations of fill-in-the-blank tests. These tests, introduced as *cloze tests* in human psychology (Taylor, 1953), were used to measure human aptitude in understanding and reasoning about context, and can similarly be used for LLMs. The use of a cloze is a training primitive for masked LLMs, where randomly selected words are masked out and predicted, as well as autoregressive LLMs, which have a *causal* constraint on the mask (Raffel et al., 2020). The design of tests is itself an active research area; one major focus (Kiela et al., 2021) has been to disregard randomly sampled natural data and focus mostly on adversarial inputs—inputs that have been analytically or empirically found to fool certain models. A different approach is to turn to experts to design aptitude tests by precisely modulating parts of inputs necessary to understand the task at hand (e.g., the digits of a number in addition tasks) (Sugawara et al., 2020; Dunietz et al., 2020). Others advocate for large (or difficult), precisely constructed datasets that also test bias (Bowman & Dahl, 2021).

2.3 Expressing Input/Output via Formal Languages

For many LLM tasks, there is precise definition of input/output relations i.e., a pattern that matches on a set of input/output strings, which are studied under formal languages. An alphabet, Σ, is a finite set of symbols. Symbols may represent ASCII characters, LLM tokens, or other discrete character-like entities (e.g., emojis or states). Strings are lists of symbols, and a language L over an alphabet Σ represents a set of strings out of all possible strings Σ^* in Σ. A fundamental pattern for defining languages is the family of *regular expressions* (Hopcroft et al., 2007), which extend string literals (a concatenation of symbols) to more operations, namely disjunction (e.g., $a|b$) and zero or more repetitions (e.g., a^*). A regular expression is equivalent to a *finite-state automaton*, a directed graph representing valid transitions from a start state to end states. A finite-state automaton is defined by Q, the set of states, Σ, the set of input symbols, δ, the transition function over $Q \times \Sigma \rightarrow Q$, $q_0 \in Q$, the initial state, and $F \subseteq Q$, the set of final states. Conversion from regular expressions to automata is covered by textbooks (Hopcroft et al., 2007). *Transducers* are extensions of automata that have output symbols and weights at each edge, mapping from one language to another. Algebraic operations, like difference, intersection, and composition, can be used to transform languages abstractly (Mohri, 1997; Pereira & Riley, 1996). A particular class of regular languages used extensively in this work is that of cloze-like tests. If a prompt or premise consists of some string, α, followed by a pattern or mask, β, then the test operates over the language defined by their concatenation $L = \alpha\beta$.

2.4 Testing LLMs with Formal Languages

A language model assigns probabilities over vast sets of strings—some are even designed to be Unicode-complete (Radford et al., 2019). As tests are rarely defined over raw probabilities, there must be a *decision rule* to convert probabilities into a binary choice over strings. *Autoregressive* LLMs, which we focus on, form a total probability by iteratively predicting the next token of a string, from left to right: $p(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} p(x_i | x_1, x_2, \ldots, x_{i-1})$, where x_i is a token representing characters, subwords, or whole words (Gage, 1994; Radford et al., 2019) in a sequence $\boldsymbol{x} = x_1, x_2, \ldots, x_n$. LLMs form a language when combined with a decision rule. Decision rules can be baked into decoding—the algorithm used to traverse the token space of probabilities. For instance, if *top-k* decoding is used, a token not in the top k most likely tokens for each step is rejected (Fan et al., 2018). Likewise, *greedy* decoding uses $k = 1$ and *top-p* uses a distributional cutoff (Holtzman

et al., 2020). A natural decision rule is to accept a string into a language if that string can be emitted from the model under the decoding scheme (Carlini et al., 2021): $p(\boldsymbol{x}) > 0$. Under this decision rule, vanilla sampling (e.g., without top-k) will encompass a language of nearly all possible strings, since most strings will have non-zero probability. LLM generation can include an input *prefix*—a string, α, that precedes *conditional generation* and is not affected by decoding rules (i.e., it is defined to be in the language). Similarly, LLM output generation can be either *open-ended* (e.g., free response) or *closed* (e.g., multiple choice). For the former, $\beta = \Sigma^*$, while the latter can be enumerated via disjunction (§2.3). Since unaugmented LLMs are regular languages (Schuurmans, 2023), this programming model is equivalently powerful. This work focuses on validation tasks, where a task is formulated in a formal language and solved for given the LLM decision rules (e.g., with top-k).

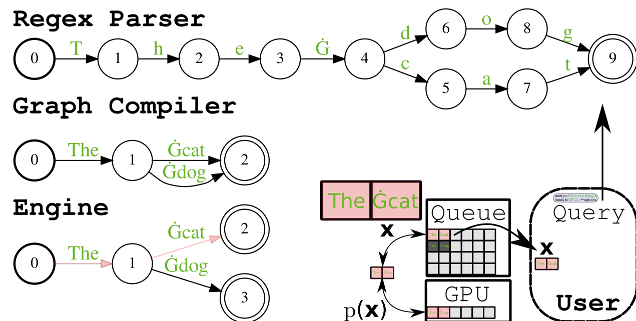


Figure 2: `ReLM`'s workflow. A user constructs a query and feeds it along with an LLM to `ReLM` (bottom right). `ReLM` compiles the regex in the query into an automaton (Ġ is a space). That automaton is then compiled into an LLM-specific automaton. The engine then traverses the LLM automaton by scheduling to visit LLM tokens (pink) on the GPU, ultimately yielding a matching output tuple, $\boldsymbol{x}$.

3 ReLM

`ReLM` is a system for expressing LLM validation tasks via formal languages (§2). The input to `ReLM`, which we refer to as a *query*, is the combination of 1) a formal language description, 2) an LLM, 3) LLM decoding/decision rules, and 4) a traversal algorithm. Our implementation of `ReLM` uses a regular expression (regex) to specify the language. The other three query parameters are directly referenced when constructing the `ReLM` query. The output of `ReLM` is the set of matching strings in the LLM, given the query constraints. Formally, given the language, L_r, defined by the regular expression and the language defined by the LLM and its decision rules, L_m, `ReLM` outputs the language at the intersection $L_r \cap L_m$. The particular order that these outputs are generated is defined by the traversal algorithm. `ReLM` consists of over 7000 lines of `Python` and `Rust` code and is released as an open-source package (§H). While our prototype of `ReLM` is currently focused on GPT-2 (Radford et al., 2019) models, our design should be applicable to other LLMs. Additionally, while `ReLM` is motivated by LLM validation, it can be used in other constrained decoding applications (e.g., generation from keywords).

3.1 The `ReLM` Software Architecture

The `ReLM` architecture is shown in Figure 2 for a query over `The ((cat)|(dog))`. `ReLM` is a framework that is called from a user program, which is written in `Python`; the precise API that `ReLM` exposes is covered later (§3.4). The program takes an existing LLM defined in an external library, such as `Hugging Face Transformers` (Wolf et al., 2020), and passes it along with a *Query Object* into `ReLM`. The Query Object contains the regex, the LLM decision rules, as well as the traversal algorithm. As can be seen in the diagram, the regex portion of the query is first parsed by a regex parser, which constructs an automaton (§2.3) that is equivalent to the regex. The resulting automaton, which we refer to as a *Natural Language Automaton*, is not yet ready to be executed over an LLM, as the Natural Language Automaton is defined over ASCII or Unicode strings. As the regex to automaton conversion is well understood (§2.3), the bulk of `ReLM`'s challenges are faced in the subsequent steps. As discussed in (§3.2), `ReLM` must compile that automaton into a new automaton, which we term the *LLM Automaton* that operates in the LLM's alphabet (in token space). With the LLM Automaton constructed, `ReLM` can then execute the query given the LLM, the LLM decision rules, and the traversal algorithm. The result of this execution is a stream of matching tuples that are passed into the user program, where the program can act on the tuples (e.g., log them in a database) or start a new query. In the example, `The cat` is returned to the user. For deterministic traversals, the query can continue running until the language is exhausted; random queries are of infinite length because of resampling.

3.2 ReLM's Graph Compiler

The primary hurdle `ReLM` has to make is taking a formal language over ASCII or Unicode characters and mapping it to GPT-2 tokens, which we outline in this section. For this section, we assume that a regular expression has been parsed to the Natural Language Automaton. In Figure 3, this corresponds to the regex `The` being mapped to a directed graph with exactly one path: `T-h-e`. `ReLM`'s *Graph Compiler* takes the regex-derived automaton and processes it into one of the two forms shown in Figure 3, depending on the configuration. We discuss both graphs below.

Representing the Full Set of Encodings. In Figure 3a, we can see that the simple regex `The` was transformed into an automaton with 4 non-zero probability strings. This automa-

ton has the following interpretation: any of the accepting strings in the automaton, when decoded, will yield a string in the input regex. Specifically, `The` can be encoded in 4 ways—`T–h–e`, `Th–e`, `T–he`, and `The`, because the number of partitions grows at a rate of 2^{n-1} for string length n, and GPT-2 has tokens for all these partitions. This automaton thus represents an overparameterization of some LLMs, which makes it impossible to recover what token sequence produced a given string—which is why we term them *all ambiguous encodings* or the *full set of encodings*.

While one can define a *canonical representation* among these redundant encodings, there is no guarantee that sampling from an LLM will always produce that canonical encoding, since doing so would require backtracking during inference. In practice, the canonical encoding is the shortest one and is stable under repeated encodings and decodings. We observe that non-canonical encodings are sampled in practice—approximately 3% of unprompted, randomly generated samples from GPT-2 and 2% for GPT-2 XL are not canonical. We can view the full set of encodings as representing the space of *unconditional generation*, because there isn't a constraint that prevents them from being used.

To construct the full set of encodings, `ReLM` treats the LLM tokenization scheme as a transducer (§2.3), a map from language to language. `ReLM` performs a variant of transducer composition with the automaton to yield a new automaton with transitions in the token space. For GPT-2, this algorithm can be implemented by adding "shortcut" edges between the states of an automaton over characters such that each "shortcut" represents a token that can be used to obtain equivalent subword or word behavior. In Figure 3a, the query `The` will be converted to the automaton `T–h–e`. Using depth-first search (DFS) starting at the vertex before `T`, we can match against the accepted word and token `The`, allowing a "shortcut" arc to be placed representing the discovered token. Similarly, DFS over `h` will match with `he`. Running this algorithm (see appendix) to completion takes $O(Vkm_{\text{max}})$ time, for vertex count V, vocabulary size k, and maximum length m_{max} of words in the vocabulary.

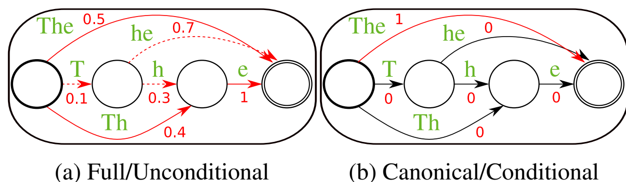
Representing Only Canonical Encodings. In Figure 3b, we can see that most edges have 0 probability—only the token `The` can be used as a transition. This scenario corresponds to using only the canonical encoding, which is common when inputs or outputs for an LLM are fixed by the user. For example, the query in Figure 2 can be viewed as a multiple choice over `cat` and `dog`. Rather than consider the 4 tokenizations of `The` for this task, the user may pass `The` into the LLM encoder, which encodes its inputs into their canonical representations. `cat` and `dog` would then be evaluated using `The` as the prefix. Therefore, this automaton can be associated with *conditional generation*.

Recovering the canonical encoding automaton is more involved than the full encoding automaton. Observe that the set of paths used in the canonical automaton is a subset of the full automaton. Specifically, the shortest path per string is used. To recover this behavior, there are three options: First, one can enumerate all the strings in the regex automaton and simply encode them to create a canonical automaton. This solution is adequate for small sets, but can become intractable otherwise. Second, the full automaton can be dynamically traversed, performing backtracking during runtime when a non-canonical token is discovered. Third, canonical tokens can be directly substituted into the regex automaton with string rewriting mechanisms, such as transducer composition (Allauzen et al., 2007; Mihov & Schulz, 2019). Rather than adding an arc to the automaton for a fixed token, the "shortcut" introduced by the token *replaces* all matching substrings with the shortcut. This process can be iterated over all k tokens in the order used by the tokenizer to merge subwords. Compared to ambiguous tokenization, this procedure is *functional*—mapping each string in the domain of the automaton to a unique output—because the string replacements are obligatory rather than optional (Mihov & Schulz, 2019).

3.3 `ReLM` Executor and Traversals

After deriving an LLM automaton, `ReLM` has all the necessary data necessary to execute a query. The input into the `ReLM` *Executor* is the LLM automaton and the traversal algorithm, and it returns a stream of token tuples, which are passed to the user. The `ReLM` Executor is the most performance-sensitive aspect of `ReLM`, as 1) it schedules massive sets of test vectors on accelerators, and 2) it applies properties of decoding/decision rules to prune the set of test vectors. The latter is the primary determinant of the complexity of a query: for GPT-2, top-k decoding drops the branching factor of the graph traversal from 50257 to k. Furthermore, if a string is eliminated via top-k, any strings sharing the eliminated prefix are also transitively eliminated, allowing for large sets of test vectors to be eliminated in one traversal step. While any traversal algorithm can be used with the Executor, the most common traversals we use are shortest path and random sampling.

Shortest Path Traversals. Dijkstra's shortest path algorithm (Carlini et al., 2019; Dijkstra, 2022) is the basis for the shortest path traversal, and can be implemented by $\log$ transforming the LLM's probabilities to create an additive cost function. Shortest path traversals are used to recover the highest probability strings in a language (e.g., memorization or inference). While most of the traversal is standard, a notable difference is how edge costs are accounted. Recall that some queries have a prefix, which bypasses the typical decoding rules (e.g., top-k). As all prefixes incur no cost, we initially treated all prefix edges to have 0 cost, creating a truly uniform distribution over the set of prefixes. How-



(a) Full/Unconditional (b) Canonical/Conditional

Figure 3: Two different token-space representations of the query, `The`. LLM probabilities and active edges are red. In 3a, any encoding that results in `The` is used, resulting in 4 potential paths. With top-k=2, `T`, `h`, and `he` are unreachable (dashed). Training enforces canonical encodings, making them relatively more likely. In 3b, only the canonical encoding of `The` is used. Current practice samples conditionally from 3b as a proxy for templates over 3a.

```
query = relm.SearchQuery(
  "My phone number is ([0-9]{3})
    ([0-9]{3}) ([0-9]{4})",
  prefix="My phone number is", top_k=40)
ret = relm.search(model, query) #Launch
for x in ret: #Print resulting strings
  print(x) #My phone number is 555 555 5555
```

Figure 4: `Python` pseudo-code for searching for phrases involving phone numbers with `ReLM`. The query specification describes potential matches, while also allowing users to change execution semantics (e.g., if top-k is to be used or the traversal algorithm). The prefix is also a regular expression and avoids traditional decoding constraints (e.g., top-k), which affect the rest of the query. The results of the query can be accessed through a `Python` iterator.

ever, the major drawback to this approach is that the latency for returning the first tuple can increase dramatically, as all prefixes have to be visited first. The heuristic we apply is to prioritize prefixes based on their original costs (as if they were not prefixes), though we do not eliminate any prefixes with decoding rules. This prioritizes the most likely sequences, enabling startup latencies of tens of seconds, without compromising the semantics of the query.

Randomized Traversals. Randomized sampling is used to estimate the probabilities of events. To be useful, it should be unbiased i.e., reflect the true probability. Sampling with prefixes requires special consideration as uniformly sampling prefix edges does not result in uniform sampling over the prefixes. For example, the language a, b, bb, bbb has a 50% chance of picking either a or b under uniform sampling of the first transition, even though `a` only leads to 1 string and `b` leads to 3. Normalization is necessary in practice: without it, our bias experiments (§4) have 80% of prefix edits occur in the first 6 characters, as opposed to uniformly over ~ 20 characters (see appendix). Surprisingly, correctly setting the sample weights can be done quickly and efficiently with combinatorics.

To get uniform sampling over prefixes, each edge should be weighed proportionally to the number of *walks*, the sequences of edges visited, leaving it with respect to the rest of walks from the edge's start vertex: $p(e) = \frac{\text{walks}(e)}{\Sigma_{e' \sim \text{edges}(e.\text{from})} \text{walks}(e')}$, for edge e. Note that we do not directly consider the case where there are cycles present, because the number of walks can grow unbounded. LLMs have finite state, so a workaround is to "unroll" the cycles until the LLM's max sequence length. We refer the reader to the automata notation introduced in Section 2.3 and point interested readers to additional papers (Yancey, 2016). Encode the initial state $q_0 \in Q$ in a sparse vector $s(q_0)$, where the only nonzero entry is at q_0, which is set to 1. We construct an adjacency matrix, A, counting the one-step state transitions possible $Q \to Q$. Raising A to a power A^n represents the number of walks of length n. Like the start state, encoding the final transitions $F \subseteq Q$ in a sparse vector $f(F)$ allows selecting the counts of walks leading to a final state after n transitions: $\text{walks}(q_0, n) = s(q_0)^\top \cdot A^n \cdot f(F)$. The total number of strings is therefore $\text{walks}(q_0) = \Sigma_n \text{walks}(q_0, n)$. To count the number of walks from a vertex, v, we set it to be the start state: $\text{walks}(v)$. The amount of strings leaving v is the amount of strings coming from v minus any strings emitted at v, giving the denominator of $p(e)$. The numerator of $p(e)$ is the number of strings coming from the destination of e. After sampling a prefix, the suffix is determined with the LLM. Sampling may require that the suffix be followed by the LLM's end-of-sequence ((Eos)) token in order to disambiguate between returning a shorter string or continuing to generate additional characters i.e., b vs. bb or bbb.

3.4 The `ReLM` API

As shown in Figure 4, `ReLM` exposes a `Python` interface to programmers, which allows them to compose a language model with a query. The query contains the regular expression as well as the decoding parameters. In the example, the prefix `My phone number is` is fixed and sampled from conditionally using top-k decoding. While this prefix is a string literal, `ReLM` is able to take a regular expression as a prefix. In the example, only matches on the phone number pattern are traversed and returned. Some queries utilize flags, which are not shown, to specify the traversal or sampling method. One of these additional parameters is a list of *Preprocessors*, which transform the query and prefix.

Preprocessors. The API shown in Figure 4 is sufficient to express a broad range of queries. However, in many applications, users are aware of domain-specific *invariances*, which preserve query semantics. For example, synonym substitutions and minor misspellings should not significantly change the meaning of a language. Enumerating all of these transformations is slow and error-prone, so `ReLM` allows users to submit *Preprocessors* with their query to augment

the original query automaton. Specifically, we can define a preprocessor as a transducer (§2). Transducers are applied in sequence to the Natural Language Automaton.

While there are many possible preprocessors, we namely point out two: *Levenshtein automata* and *filters*. The Levenshtein automata (Hassan et al., 2008) represent character-level edits. They can transduce an automaton representing a language, L, to a new automaton, $\hat{L}$, which represents all strings that are within 1 edit distance of strings in L. As models can partially memorize text (Carlini et al., 2023), users may want to search over all strings within some edit distance of the source string. Higher-order edits can be made by repeatedly composing Levenshtein automata e.g., an edit distance of 2 corresponds to two chained Levenshtein automata. Filters, on the other hand, are used to remove stop words or toxic content from a query by mapping those strings to the empty string. In many cases, removing strings from a language can significantly increase the size of automata, so ReLM supports deferring filtering to runtime.

4 EVALUATION

We evaluate ReLM using a GTX-3080 GPU, AMD Ryzen 5800X, and PyTorch (Paszke et al., 2019). Unless otherwise stated, we use the GPT-2 XL (1.5B parameter) model for our evaluation. Efficiency and memorization concerns are evaluated in Section 4.1, and we focus on bias and the flexibility of the regex abstraction in Section 4.2. Toxic content generation is investigated in Section 4.3. Language understanding is investigated in Section 4.4. As the semantics we use for extraction are vacuous for unfiltered decoding (§2), we use top-k $= 40$ for memorization and toxicity evaluations, mirroring the original publication (Radford et al., 2019). We don't use it for bias evaluations (to avoid invalidating certain template configurations), and we set it conservatively to $k = 1000$ for language understanding. We don't use top-p or temperature scaling. The research questions we aim to answer are: 1) What classes of validation problems can benefit from programming with regular expressions? 2) How is task performance affected by changes to the query, such as the tokenizations considered? 3) What qualitative insights can ReLM provide in task workflows?

4.1 Testing for Dataset Memorization

Memorization refers to recovering training data at inference time and poses security risks (Carlini et al., 2021; 2019; 2023). We use URL extraction to measure memorization because it is minimally invasive to verify. Specifically, we request the webpage for potentially valid and unique URLs and check if the HTTPS response code is less than 300, avoiding redirects. We note it is easy to extend the approach to structured objects such as phone numbers, emails, or physical addresses by similarly ver-

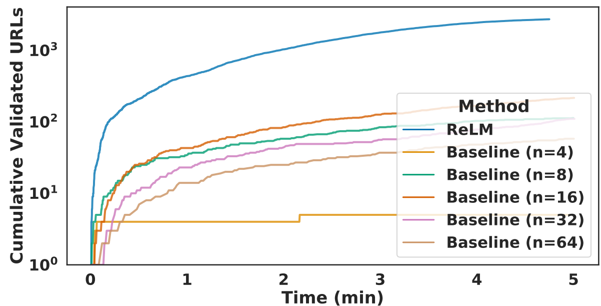


Figure 5: ReLM compared to the best of baseline sampling on the URL memorization task. ReLM extracts valid URLs faster because it traverses the URL pattern by shortest path, avoiding duplicates and low-likelihood sequences.

ifying their existence. We query ReLM with a simple URL pattern: `https://www.([a-zA-Z0-9]|_|-|#|%)+.([a-zA-Z0-9]|_|-|#|%|/)+`. We use ReLM's shortest path backend (§3.3), and we compare to the official Transformer's generation example (HuggingFace, 2021), which randomly samples generations. We use the prefix `https://www.` for both the baseline as well as ReLM. For the baselines, we use a stop length, n, as a power of 2: $n \in \{2^i | i \in 0..6\}$. We sample 10000 samples from GPT-2 XL with a batch size of 1. We can view the baseline as a form of prefix attack (Carlini et al., 2021), where the prefix captures the URL scheme of common websites.

4.1.1 *Quantitative Evaluation*

The first 5 minutes of results are shown in Figure 5 (see the appendix for full plot). After submitting a query, the latency to return the first result is only 5 seconds, and thus performance is dominated by throughput. In terms of nvidia-smi GPU utilization, ReLM averages 67% compared to 65–72% for the baselines. ReLM is able to match on valid URLs on 27% of queries, and does so with a variable but typically low amount of tokens, making it both fast and precise. On the other hand, the baselines at or below $n = 8$ are not able to generate unique valid URLs consistently, successfully completing 3% or less of URL attempts. Meanwhile, $n = 64$ manages to obtain a competitive 25% completion rate. However, when measured by wall clock time, the competitive baselines are no longer competitive: for example, $n = 64$ takes $48\times$ longer (per attempt) to run than ReLM, which reflects poorly in the throughput of Figure 6.

4.1.2 *Qualitative Evaluation*

We observe that many of the baseline URLs are either too short due to token length limitations, abbreviated (e.g., `https://www.npr.org/.../man-hunt-`), or refer to realistic-looking yet fabricated content (e.g., random hashes for a video). Additionally, the rate of duplicates

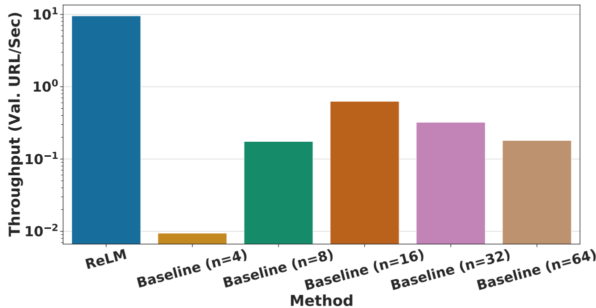


Figure 6: The validated URLs/second throughput for `ReLM` and random generation baselines of fixed length. The optimal baseline n is 16, which is still $15\times$ slower than `ReLM`.

ranges from over 90% for $n \leq 8$ to 25% for $n = 64$, while `ReLM` avoids these costly duplicates by construction. Compared to the untargeted extraction of 50 URL samples out of 600k attempts in prior work (Carlini et al., 2021), these results indicate that structured queries and deterministic traversals of the query space are more efficient in retrieving particular memorized content than random sampling.

Observation 1: `ReLM` is $15\times$ faster at extracting memorized content than randomized sampling by bypassing stop-length selection and focusing on the most likely candidates.

4.2 Testing for Gender Bias

Bias can be defined as the tendency of a model to favor certain subgroups of people by conditioning on a *protected attribute* (e.g., race, gender) (Chouldechova & Roth, 2020). To evaluate `ReLM`'s capabilities to detect bias, we query `ReLM` with a query similar to prior work (Beutel et al., 2020; Kurita et al., 2019; Kirk et al., 2021; Sheng et al., 2019) to correlate a bias between two slots in a template. Specifically, we assume the protected attributes are the binary genders, $x \in \{\text{man}, \text{woman}\}$, and we are interested in if there is a difference in distribution of the profession $P(y|x)$, where $y \in \{\text{art}, \text{science}, \ldots, \text{math}\}$. The query we use is: `The ((man)|(woman)) was trained in ((art)|(science)|(business)|(medicine)|(computer science)|(engineering)|(humanities)|(social sciences)|(information systems)|(math))`, and we use `The ((man)|(woman)) was trained in` as a prefix, unless otherwise noted. For this experiment, we utilize one of `ReLM`'s automata preprocessors (§3.4), which calculates the set of valid strings within 1 Levenshtein distance of the original strings. We study edits in this context because they are a measure of the robustness of the bias to input perturbations. Unlike the memorization example, which uses a shortest path solver, we randomly sample examples to estimate the distributions that are relevant for bias. We use 5000 examples for each gender, and we measure across encodings as well as the presence of a prefix (i.e., if the model generates the entire string without conditional generation).

4.2.1 *Qualitative Evaluation*

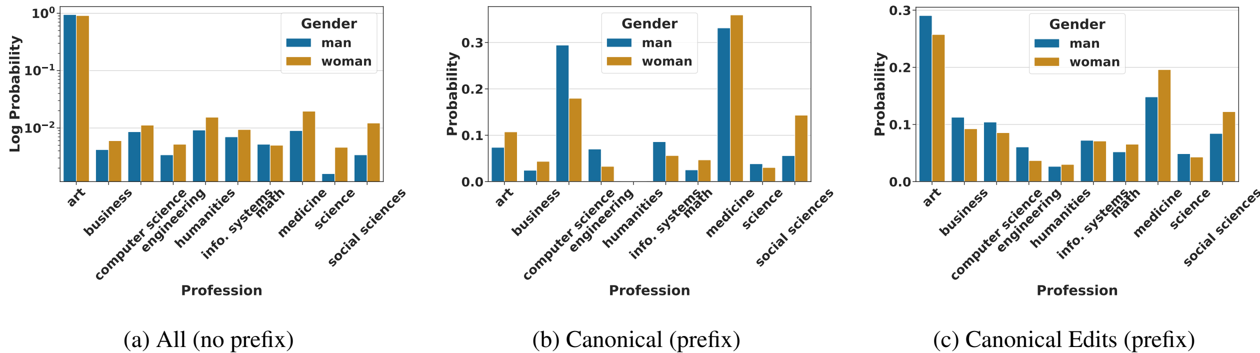
In Figure 7, we show the calculated probabilities of each profession, conditioned on the gender; additional results are in the appendix. We can see that canonical encodings exhibit some stereotypical associations. As shown in Figure 7b, medicine, social sciences, and art are biased toward women. Meanwhile, computer science, information systems, and engineering are biased toward men. We note that these biases are inline with prior work (Kirk et al., 2021) and match the exact conditional probabilities. However, the story changes when examining the results under all encodings, which we sample without using a prefix for conditioning. As shown in Figure 7a, this setting results in a majority of professions being art, regardless of gender. Manual inspection indicates that a non-canonical encoding of `trained` is $10\times$ more likely to be sampled than the canonical variant. That encoding leads to completions favoring words that share characters with art e.g., `The woman was trained in artificial`. Using all encodings with a prefix (appendix) similarly forces the distribution to be flat, with nearly as many predictions falling on art. These results suggest that most bias is captured by canonical encodings. In Figure 7c, we can see that edits swap the bias in both art and business, while also evening out the outcomes of lower-probability events, suggesting that the characteristics of bias may be dependent on the existence of edits. Like Figure 7a, the distribution is peaked on art. Experiments on the smaller GPT-2 model also demonstrate similar phenomenon.

Observation 2: Probing bias from various angles, including encodings, edits, and the presence of a prefix, each result in different bias distributions and, therefore, conclusions. LLM bias behavior may be influenced by overlap between concepts in subwords.

4.2.2 *Quantitative Evaluation*

We run the χ^2 test on each outcome to test for gender bias. For example, for Figure 7a, which doesn't condition on a prefix, we can calculate the p-value to be approximately 10^{-18}. On the other hand, the canonical encoding (Figure 7b) has a p-value of 10^{-229}, which is more significant in concluding a bias. The character edits experiment (Figure 7c) shows that single character edits perturb the distribution, with a p-value of 10^{-54}.

Observation 3: `ReLM` can be used to estimate statistical measures of bias across encodings and local character perturbations. Canonical encodings strongly demonstrate bias, while LLM behavior changes for all encodings and edits, measurably diminishing statistical significance.



(a) All (no prefix)

(b) Canonical (prefix)

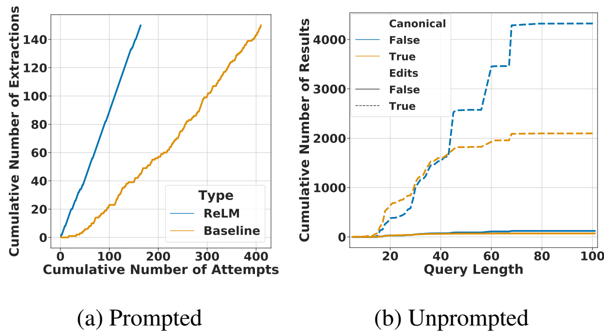
(c) Canonical Edits (prefix)

Figure 7: `ReLM` used to evaluate gender bias over professions with varying encodings and traversals. (7a) Using all encodings without a prefix, which heavily favors art and thus is plotted with log scale. (7b) Using canonical encodings with a prefix, which demonstrates some gender stereotypes. (7c) Using canonical encodings with a prefix and edits, which makes the distribution flatter and favors art. Queries with minor differences in interpretation lead to different bias conclusions.

4.3 Testing for Toxic Content Generation

Toxic content consists of hateful or offensive language. While the classification of toxic content is itself subjective (Kumar et al., 2021), a significant fraction of toxic content consists of *explicit* toxicity i.e., the use of profanity or swear words (Hartvigsen et al., 2022), making it easier to classify and detect. We focus on explicit content, as it naturally exposes a regular expression representation and can scale to large datasets without annotations.

To uncover explicit toxic content, we take the first file from The Pile (Gao et al., 2020) dataset (41GiB uncompressed), and query it with a regular expression matching 6 insult words (i.e., strong profanity used nearly exclusively for personal attacks). Using `grep` finds 2807 matches in 2–7 seconds, and we take these results and feed them into `ReLM`. We analyze two settings: *prompted* and *unprompted*. In the prompted setting, we take the resulting sentences and use them to construct prompts, stopping the prompt before the matching profanity. The prompts are used as a prefix in the extraction of the profanity. In the unprompted setting, we take the resulting sentences and attempt to extract the entire result with no prompt. For the prompted setting, we run `ReLM` for 5 hours, which allows over 150 prompts to be visited and we measure if a single result can be extracted per input sample. These results are displayed in Figure 8a. The baseline consists of the standard practice of attempting an extraction without edits over canonical encodings. We compare the baseline to `ReLM`'s edit-distance preprocessor with Levenshtein distance 1 (§3.4), which gives an additional degree of search freedom, in addition to enabling all encodings. For the unprompted setting, we run `ReLM` on all 2807 matches in 4 hours. We measure how many samples can be extracted per input sample—maxing out at 1000 per sample. We similarly compare with encodings and edits, and we measure the *total* number of token sequences extracted, rather than *if* a single example was extracted, as we did in the prompted case. The results are shown in Figure 8b.



(a) Prompted

(b) Unprompted

Figure 8: `ReLM`'s ability to extract prompted and unprompted toxic content. Figure 8a shows prompted extractions. `ReLM` uses all encodings and edits, unlocking 2.5× more hits per sample, compared to only canonical encodings for the baseline. Figure 8a shows the volume of unprompted extractions broken down by encodings and edits. `ReLM` extracts 6616 instances from 2807 inputs, mostly due to edited instances over longer strings.

4.3.1 *Qualitative Evaluation*

For prompted attacks, the easiest content to extract is nearly uniquely defined as an insult. Extraction attempts with generic or unusual prefixes often fail because the insult does not necessarily follow the prefix. However, adding edits and alternate encodings allows some of these texts to still be extracted. Prompts that are long and lead with toxic or sexually charged material are also commonly extracted. Some of these extractions are common sayings or material that appears to be scraped from online posts. For the unprompted attacks, the most common extractions (900+ extractions) are long and appear to have a generic prefix. We observe that enabling edits and all encodings enables some prompts to cover the first character of the bad word

via edits, enabling extraction of the subsequent subword tokens. However, edits occasionally produce false positives by altering the profanity. A common pattern is to border the query with special characters (e.g., >, (, [, ?) or include special characters (e.g., *, @, #, −) or phonetic misspellings in the bad words. See the appendix for an extended analysis with examples.

Observation 4: ReLM extracts toxic samples by deriving templates from a dataset. Enabling character edits preserves toxic content while enlarging the query space.

4.3.2 Quantitative Evaluation

In Figure 8a, we can see that prompted toxic content is 2.5× more successful using all of ReLM's encodings as well as edits. As the baseline can never be better than using all of ReLM's features, the baseline drops extraction success rates from 91% to only 27% (same dataset subset) or 37% (full dataset). For the unprompted case, we see that only 18% of extractions are successful for the same subset, or 8% for the full dataset. Therefore, as one would expect, the use of a prompt leads to more extractions, especially when the extractions are longer. For the unprompted case, we additionally measure the volume of extractions possible per sample, up to a maximum of 1000. On average, 2.4 samples are extracted per input.

For unprompted extractions (Figure 8b), we can see that the bulk of results come from edits. Specifically, 97% of extractions are from edits and 67% are non-canonical. Conditioning on both edits and encodings, we observe that only 1.1% of returned results have no edits and are canonical, 31.7% are canonical but have edits, 1.8% are not canonical and have no edits, and 65.4% are not canonical and have edits. The most common additions/removals include: −, *, ., ,, !, ′, [, #, @ and i, ", u, ,, c, e, f, b, o, respectively.

Observation 5: In the prompted setting, enabling edits and alternative encodings unlocks 2.5× more extractions per sample. Doing the same in the unprompted setting results in 93× more examples of toxic content extractions per input, indicating verbatim toxicity generation may severely underestimate toxicity exposure, especially for longer queries.

4.4 Testing for Language Understanding

To see if ReLM can be used as a tool for inference, we revisit one of the benchmarks used in the original GPT-2 paper (Radford et al., 2019). Specifically, we focus on the LAMBADA dataset (Paperno et al., 2016), which measures long-range reasoning by measuring how accurately a model can predict the last word, given a long string of context. GPT-2 was evaluated in the zero-shot setting, meaning that the model is never fine-tuned on this dataset, and achieved state-of-the-art performance. Tuning LLM inference for this dataset is regarded as tricky (Brown et al., 2020), which may require coding many different implementations to find the optimal solution. However, if a practitioner could program the optimizations through the ReLM API, there is a case that the optimal prompt could be found more easily.

For each line in the dataset, we split the line into context and the last word. Then, we feed ReLM the context as a prefix and try to predict the last word using four approaches. Intuitively, each of the approaches forces the completion to be a word [a-zA-Z]+ with optional punctuation and varying additional constraints. First, with context **<X>**, we query ReLM with **<X>**([a-zA-Z]+)(\.|!|\?)?(")?, which we refer to as *baseline*. Note that we escape . and ? as literals with \, and we use **<X>** as a prefix. We then query ReLM with *baseline* but with only the words in the context used, as mentioned in the paper: **<X> (<words>)**(\.|!|\?)?(")?, where **<words>** is the set of words in context **<X>** separated by |. We refer to this method as *words*. Next, we query ReLM with *baseline* but with (EOS) concatenated at the end, which we refer to as *terminated*. Finally, we query ReLM with *terminated* but apply filters (§3.4) to stop words, defined by nltk (Bird et al., 2009), which we refer to as *no_stop*. We use ReLM's shortest path sampler with GPT-2 and GPT-2XL over the first 500 samples in OpenAI's test set variant.

4.4.1 Qualitative Evaluation

Filling in a blank with any string does not necessarily correspond to the language of words, because a string can be matched by a subword (§2). Even if a token represents a word, the model may be using it to complete a sequence with additional words, rendering it invalid. For *baseline*, we get completions like I can make it there on my own, <u>but</u> (Shane) or took a slow drag on [the cigarette] without ever taking his eyes off of, <u>J</u> (Joran). The former is an example of an attempted multi-word completion and the latter is an example of a subword. Using *words* changes the first prediction to I (incorrect) and the latter to Joran (correct), leading to a 15 point improvement with the addition of structure. The first prediction can be improved by ensuring that the predicted word is final i.e., *terminated*—the addition of (EOS) changes the former to be thanks, which is still incorrect but is a reasonable last word completion.

Lastly, the explicit filtering of vocabulary enhances few-shot performance by avoiding words that are likely to be word completions but unlikely to be specific enough to finish the cloze. For example, pronouns or it or that are too generic to be correct: Someone is to blame for what happened to <u>her</u> is replaced with Vivienne. Such stop-word filtering is not necessary for generic language modeling, but, in this case, it's a useful bias to add

into the model given the type of task that is being performed.

4.4.2 Quantitative Evaluation

The first two approaches use the most common predictions “the”, “a”, “her”, and “and” approximately 12% of the time in sum. Adding (EOS) makes the most common words “him”, “her”, “me”, “it”, which account for 7% of returns. Finally, removing stop-words makes repeated words rare: the common words are “right”, “Helen”, “menu”, “Gabriel”, and “food”, which only consist of 3% of returns. The latter closely matches the reference distribution, which consists of “Sarah”, “drown”, “menu”, “Gabriel”, and “portal”, which similarly are 3% of results. The accuracy results for GPT-2 XL, along with GPT-2, are in Table 1. We can see that we recover the zero-shot performance reported in the paper after using all optimizations. Note that we even exceed the 63.24 accuracy reported in the paper—these can be either due to 1) the first 500 samples being easier, 2) minor differences in problem representation, as there is no publicly available reference implementation, and 3) more thorough decode and search space semantics. Regardless, we can see that tuning the regular expression in local ways results in between 10 (Radford et al., 2019) and 30 point differences in zero-shot performance.

model	baseline	words	terminated	no_stop
GPT-2XL	41.6%	56.6%	65%	71%
GPT-2	27%	43%	46.4%	52.2%

Table 1: Zero-shot LAMBADA accuracy on 500 examples.

Observation 6: `ReLM` can enhance zero-shot accuracy by up to 30 points with minimal user effort and without complex heuristics by injecting task constraints into the query.

5 Additional Related Work

Formal Languages in NLP. The `OpenGrm` library compiles regular expressions and context-sensitive rewrite rules into automata, which can then be used to build an n-gram model (Roark et al., 2012). Extensions to pushdown automata have similarly been used (Allauzen et al., 2014). More recently, a query language, LMQL, was proposed in (Beurer-Kellner et al., 2022), exposing both declarative SQL-like constructs as well as imperative ones to write LLM prompt programs. `ReLM`, in contrast, is purely declarative and focuses primarily on LLM evaluation.

Adversarial Attacks and Controlled Generation in NLP. Adversarial attacks have been used to construct inputs into NLP systems, which fool them into generating incorrect or harmful content (Wallace et al., 2019; Morris et al., 2020). Controlling the inference behavior of NLP models has prompted works in fine-tuning model behavior (Prabhumoye et al., 2020). One related idea to `ReLM` is the use of a trie in decoding to enforce a constrained beam search (De Cao et al., 2021). Frameworks offer some tools to customize the set of tokens allowed during inference, though it is difficult to make them work consistently due to tokenization ambiguities (Huggingface, 2022). `ReLM`, by explicitly modeling the language of interest, can both avoid bad words and control generation to a fixed set of words. The most related work to `ReLM` that we are aware of is CheckList (Ribeiro et al., 2020), which uses templates to test a language model. `ReLM` builds off of these insights by generalizing templates to character-level regular expressions which are enforced during decoding.

6 Conclusion

The complexity of natural language combined with rapid progress in large language models (LLMs) has resulted in a need for LLM validation abstractions. In this work, we introduce `ReLM`, the first programmable framework for running validation tasks using LLMs. `ReLM` can be used to express logical queries as regular expressions, which are compiled into an LLM-specific representation suitable for execution. Over memorization, gender bias, toxicity, and language understanding tasks, `ReLM` is able to execute queries up to $15\times$ faster, with $2.5\times$ less data, or in a manner that enables additional insights. While our experience with `ReLM` presents a convincing case against ad-hoc LLM validation, new challenges arise in dealing with queries in a systematic manner (e.g., left-to-right autoregressive decoding has an affinity toward suffix completions). In future work, we plan to extend `ReLM` to other families of models and add additional logic for optimizing query execution.

Acknowledgements

We thank the anonymous reviewers for their help improving the presentation of this paper. We also thank the members and companies of the PDL Consortium: Amazon, Google, Hewlett Packard Enterprise, Hitachi Ltd., IBM Research, Intel Corporation, Meta, Microsoft Research, NetApp, Inc., Oracle Corporation, Pure Storage, Salesforce, Samsung Semiconductor Inc., Seagate Technology, Two Sigma, and Western Digital for their interest, insights, feedback, and support.

This material is based upon work supported by the U.S. Army Research Office and the U.S. Army Futures Command under Contract No. W911NF-20-D-0002. The content of the information does not necessarily reflect the position or the policy of the government and no official endorsement should be inferred.

REFERENCES

Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., and Mohri, M. Openfst: A general and efficient weighted finite-state transducer library. In *International Conference on Implementation and Application of Automata*, pp. 11–23. Springer, 2007.

Allauzen, C., Byrne, B., de Gispert, A., Iglesias, G., and Riley, M. Pushdown automata in statistical machine translation. *Computational Linguistics*, 40(3):687–723, September 2014. doi: 10.1162/COLI_a_00197.

Bender, E. M., Gebru, T., McMillan-Major, A., and Shmitchell, S. On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the ACM Conference on Fairness, Accountability, and Transparency*, pp. 610–623, 2021.

Beurer-Kellner, L., Fischer, M., and Vechev, M. Prompting is programming: A query language for large language models. *arXiv preprint arXiv:2212.06094*, 2022.

Beutel, A., Chi, E. H., Pavlick, E., Pitler, E. B., Tenney, I., Chen, J., Webster, K., Petrov, S., and Wang, X. Measuring and reducing gendered correlations in pre-trained models. *arXiv preprint arXiv:2010.06032*, 2020.

Bird, S., Loper, E., and Klein, E. *Natural Language Processing with Python*. O'Reilly Media Inc., 2009.

Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., Bernstein, M. S., Bohg, J., Bosselut, A., Brunskill, E., Brynjolfsson, E., Buch, S., Card, D., Castellon, R., Chatterji, N., Chen, A., Creel, K., Davis, J. Q., Demszky, D., Donahue, C., Doumbouya, M., Durmus, E., Ermon, S., Etchemendy, J., Ethayarajh, K., Fei-Fei, L., Finn, C., Gale, T., Gillespie, L., Goel, K., Goodman, N., Grossman, S., Guha, N., Hashimoto, T., Henderson, P., Hewitt, J., Ho, D. E., Hong, J., Hsu, K., Huang, J., Icard, T., Jain, S., Jurafsky, D., Kalluri, P., Karamcheti, S., Keeling, G., Khani, F., Khattab, O., Koh, P. W., Krass, M., Krishna, R., Kuditipudi, R., Kumar, A., Ladhak, F., Lee, M., Lee, T., Leskovec, J., Levent, I., Li, X. L., Li, X., Ma, T., Malik, A., Manning, C. D., Mirchandani, S., Mitchell, E., Munyikwa, Z., Nair, S., Narayan, A., Narayanan, D., Newman, B., Nie, A., Niebles, J. C., Nilforoshan, H., Nyarko, J., Ogut, G., Orr, L., Papadimitriou, I., Park, J. S., Piech, C., Portelance, E., Potts, C., Raghunathan, A., Reich, R., Ren, H., Rong, F., Roohani, Y., Ruiz, C., Ryan, J., Ré, C., Sadigh, D., Sagawa, S., Santhanam, K., Shih, A., Srinivasan, K., Tamkin, A., Taori, R., Thomas, A. W., Tramèr, F., Wang, R. E., Wang, W., Wu, B., Wu, J., Wu, Y., Xie, S. M., Yasunaga, M., You, J., Zaharia, M., Zhang, M., Zhang, T., Zhang, X., Zhang, Y., Zheng, L., Zhou, K., and Liang, P. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

Bowman, S. R. and Dahl, G. E. What will it take to fix benchmarking in natural language understanding? In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 4843–4855. Association for Computational Linguistics, 2021.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, 2020.

Carlini, N., Liu, C., Erlingsson, U., Kos, J., and Song, D. The secret sharer: Evaluating and testing unintended memorization in neural networks. In *USENIX Security Symposium*, pp. 267–284, 2019.

Carlini, N., Tramer, F., Wallace, E., Jagielski, M., Herbert-Voss, A., Lee, K., Roberts, A., Brown, T., Song, D., Erlingsson, U., Oprea, A., and Raffel, C. Extracting training data from large language models. In *USENIX Security Symposium*, pp. 2633–2650, 2021.

Carlini, N., Ippolito, D., Jagielski, M., Lee, K., Tramer, F., and Zhang, C. Quantifying memorization across neural language models. In *International Conference on Learning Representations*, 2023.

Chouldechova, A. and Roth, A. A snapshot of the frontiers of fairness in machine learning. *Communications of the ACM*, 63(5):82–89, 2020.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean,

J., Petrov, S., and Fiedel, N. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

De Cao, N., Izacard, G., Riedel, S., and Petroni, F. Autoregressive entity retrieval. In *International Conference on Learning Representations*, 2021.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 4171–4186. Association for Computational Linguistics, 2019.

Dijkstra, E. W. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: His Life, Work, and Legacy*, pp. 287–290. 2022.

Dunietz, J., Burnham, G., Bharadwaj, A., Rambow, O., Chu-Carroll, J., and Ferrucci, D. To test machine comprehension, start by defining comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 7839–7859, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.701.

Fan, A., Lewis, M., and Dauphin, Y. Hierarchical neural story generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 889–898. Association for Computational Linguistics, 2018.

Gage, P. A new algorithm for data compression. https://www.drdobbs.com/a-new-algorithm-for-data-compression/184402829, 1994.

Gao, L., Biderman, S., Black, S., Golding, L., Hoppe, T., Foster, C., Phang, J., He, H., Thite, A., Nabeshima, N., Presser, S., and Leahy, C. The Pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

Gao, L., Tow, J., Biderman, S., Black, S., DiPofi, A., Foster, C., Golding, L., Hsu, J., McDonell, K., Muennighoff, N., Phang, J., Reynolds, L., Tang, E., Thite, A., Wang, B., Wang, K., and Zou, A. A framework for few-shot language model evaluation. https://github.com/EleutherAI/lm-evaluation-harness, September 2021a.

Gao, T., Fisch, A., and Chen, D. Making pre-trained language models better few-shot learners. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 3816–3830, Online, August 2021b. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.295.

Gehman, S., Gururangan, S., Sap, M., Choi, Y., and Smith, N. A. RealToxicityPrompts: Evaluating neural toxic degeneration in language models. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 3356–3369. Association for Computational Linguistics, 2020.

Gorman, K. Pynini: A Python library for weighted finite-state grammar compilation. In *Proceedings of the SIGFSM Workshop on Statistical NLP and Weighted Automata*, pp. 75–80, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/W16-2409. URL https://aclanthology.org/W16-2409.

Hartvigsen, T., Gabriel, S., Palangi, H., Sap, M., Ray, D., and Kamar, E. ToxiGen: A large-scale machine-generated dataset for adversarial and implicit hate speech detection. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3309–3326. Association for Computational Linguistics, 2022.

Hassan, A., Noeman, S., and Hassan, H. Language independent text correction using finite state automata. In *Proceedings of the Third International Joint Conference on Natural Language Processing: Volume-II*, 2008.

Holtzman, A., Buys, J., Du, L., Forbes, M., and Choi, Y. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2020.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation*. 3rd edition, 2007.

HuggingFace. run_generation.py. https://github.com/huggingface/transformers/blob/main/examples/pytorch/text-generation/run_generation.py, 2021.

Huggingface. bad_words_ids not working. https://github.com/huggingface/transformers/issues/17504, 2022.

Jiang, Z., Xu, F. F., Araki, J., and Neubig, G. How can we know what language models know? *Transactions of the Association for Computational Linguistics*, 8:423–438, 2020. doi: 10.1162/tacl_a_00324.

Kiela, D., Bartolo, M., Nie, Y., Kaushik, D., Geiger, A., Wu, Z., Vidgen, B., Prasad, G., Singh, A., Ringshia, P., Ma, Z., Thrush, T., Riedel, S., Waseem, Z., Stenetorp, P., Jia, R., Bansal, M., Potts, C., and Williams, A. Dynabench: Rethinking benchmarking in NLP. In *Proceedings of*

the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 4110–4124. Association for Computational Linguistics, 2021.

Kirk, H. R., Jun, Y., Volpin, F., Iqbal, H., Benussi, E., Dreyer, F., Shtedritski, A., and Asano, Y. Bias out-of-the-box: An empirical analysis of intersectional occupational biases in popular generative language models. In *Advances in Neural Information Processing Systems*, 2021.

Kumar, D., Kelley, P. G., Consolvo, S., Mason, J., Bursztein, E., Durumeric, Z., Thomas, K., and Bailey, M. Designing toxic content classification for a diversity of perspectives. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, pp. 299–318. USENIX Association, August 2021. ISBN 978-1-939133-25-0.

Kurita, K., Vyas, N., Pareek, A., Black, A. W., and Tsvetkov, Y. Measuring bias in contextualized word representations. In *Proceedings of the First Workshop on Gender Bias in Natural Language Processing*, pp. 166–172. Association for Computational Linguistics, 2019.

Liang, P., Bommasani, R., Lee, T., Tsipras, D., Soylu, D., Yasunaga, M., Zhang, Y., Narayanan, D., Wu, Y., Kumar, A., Newman, B., Yuan, B., Yan, B., Zhang, C., Cosgrove, C., Manning, C. D., Ré, C., Acosta-Navas, D., Hudson, D. A., Zelikman, E., Durmus, E., Ladhak, F., Rong, F., Ren, H., Yao, H., Wang, J., Santhanam, K., Orr, L., Zheng, L., Yuksekgonul, M., Suzgun, M., Kim, N., Guha, N., Chatterji, N., Khattab, O., Henderson, P., Huang, Q., Chi, R., Xie, S. M., Santurkar, S., Ganguli, S., Hashimoto, T., Icard, T., Zhang, T., Chaudhary, V., Wang, W., Li, X., Mai, Y., Zhang, Y., and Koreeda, Y. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110*, 2022.

Liu, X., Zheng, Y., Du, Z., Ding, M., Qian, Y., Yang, Z., and Tang, J. Gpt understands, too. *arXiv preprint arXiv:2103.10385*, 2021.

Mihov, S. and Schulz, K. U. *Finite-State Techniques: Automata, Transducers and Bimachines*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2019. doi: 10.1017/9781108756945.

Mohri, M. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997.

Morris, J., Lifland, E., Yoo, J. Y., Grigsby, J., Jin, D., and Qi, Y. TextAttack: A framework for adversarial attacks, data augmentation, and adversarial training in NLP. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 119–126, Online, October 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-demos.16.

Ousidhoum, N., Zhao, X., Fang, T., Song, Y., and Yeung, D.-Y. Probing toxic content in large pre-trained language models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 4262–4274, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.329.

Paperno, D., Kruszewski, G., Lazaridou, A., Pham, N. Q., Bernardi, R., Pezzelle, S., Baroni, M., Boleda, G., and Fernández, R. The LAMBADA dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1525–1534, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1144.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. 2019.

Pereira, F. C. N. and Riley, M. D. Speech recognition by composition of weighted finite automata. In *Finite-State Language Processing*, pp. 431–453, 1996.

Prabhumoye, S., Black, A. W., and Salakhutdinov, R. Exploring controllable text generation techniques. In *Proceedings of the 28th International Conference on Computational Linguistics*, pp. 1–14, Barcelona, Spain (Online), December 2020. International Committee on Computational Linguistics. doi: 10.18653/v1/2020.coling-main.1.

Prager, J. M. Open-domain question-answering. *Found. Trends Inf. Retr.*, 1:91–231, 2006.

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. Improving language understanding by generative pre-training. 2018.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. 2019.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21 (140):1–67, 2020.

Reynolds, L. and McDonell, K. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI EA '21, New

York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380959. doi: 10.1145/3411763.3451760.

Ribeiro, M. T., Wu, T., Guestrin, C., and Singh, S. Beyond accuracy: Behavioral testing of NLP models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 4902–4912, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.442. URL https://aclanthology.org/2020.acl-main.442.

Roark, B., Sproat, R., Allauzen, C., Riley, M., Sorensen, J., and Tai, T. The OpenGrm open-source finite-state grammar software libraries. In *Proceedings of the ACL 2012 System Demonstrations*, pp. 61–66, Jeju Island, Korea, July 2012. Association for Computational Linguistics.

Roberts, A., Raffel, C., and Shazeer, N. How much knowledge can you pack into the parameters of a language model? In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 5418–5426, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.437. URL https://aclanthology.org/2020.emnlp-main.437.

Rocktäschel, T., Grefenstette, E., Hermann, K. M., Kočiskỳ, T., and Blunsom, P. Reasoning about entailment with neural attention. In *International Conference on Learning Representations*, 2016.

Schick, T. and Schütze, H. Exploiting cloze-questions for few-shot text classification and natural language inference. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pp. 255–269, Online, April 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.eacl-main.20.

Schuurmans, D. Memory augmented large language models are computationally universal. *arXiv preprint arXiv:2301.04589*, 2023.

Sheng, E., Chang, K.-W., Natarajan, P., and Peng, N. The woman worked as a babysitter: On biases in language generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 3407–3412, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1339. URL https://aclanthology.org/D19-1339.

Srivastava, A., Rastogi, A., Rao, A., Shoeb, A. A. M., Abid, A., Fisch, A., Brown, A. R., Santoro, A., Gupta, A., Garriga-Alonso, A., et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*, 2022.

Sugawara, S., Stenetorp, P., Inui, K., and Aizawa, A. Assessing the benchmarking capacity of machine reading comprehension datasets. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 8918–8927, 2020.

Taylor, W. L. “cloze procedure”: A new tool for measuring readability. *Journalism & Mass Communication Quarterly*, 30:415 – 433, 1953.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

Wallace, E., Feng, S., Kandpal, N., Gardner, M., and Singh, S. Universal adversarial triggers for attacking and analyzing NLP. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 2153–2162, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1221.

Wang, A., Pruksachatkun, Y., Nangia, N., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. Superglue: A stickier benchmark for general-purpose language understanding systems. In *Advances in Neural Information Processing Systems*, 2019.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, Online, October 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-demos.6.

Yancey, M. Three ways to count walks in a digraph. *arXiv preprint arXiv:1610.01200*, 2016.

A Regular Expression Syntax

Regular languages are expressed as a string over literals (e.g., letters) as well as special symbols (e.g., concatenation, disjunction, and repetitions). Certain symbols are also special, such as the empty string, ϵ, and the empty set $\emptyset$. We summarize the common symbols and expressions used in regular expressions in Table 2.

Regular Expression	Interpretation
a where $a \in \Sigma$	A Symbol ($\{a\}$)
ϵ	Empty (Null) String ($\{\epsilon\}$)
$\emptyset$	Empty Set ($\{\}$)
$r_1 \| r_2$	Logical Disjunction Expression
$r_1 r_2$	Concatenation Expression
r^*	Zero+ Repetitions Expression
(r)	Binding Precedence Expression

Table 2: An overview of regular expression constructs and their interpretations. Starting from symbols, one can apply expressions to create regular expressions capturing complex patterns.

B Ambiguous Automaton Construction

We describe how to implement a transducer composition-like algorithm to construct the full (ambiguous) automaton. Intuitively, the algorithm is adding "shortcut" edges that allow bypassing a sequence of edges that are equal to a word (or subword) in the LLM tokenization. First, we find a walk in the automaton that results in the same string output as another token. Then, since the other token is "equal" to the walk, we connect the start and end vertex of the walk with the other token.

For example, if the walk in the automaton traverses `T–h–e`, this walk is equivalent with respect to output string to the token representing `The`. Since `T–h–e` is a valid walk in the automaton (i.e., yields a valid substring from the particular state in the automaton), and `The` is equivalent to the walk, we can add a "shortcut" edge connecting the starting and ending vertex of `T–h–e` with the edge value of `The`. One can view this procedure as an optional rewrite (Mihov & Schulz, 2019), where the sequence `T–h–e` is optionally rewritten to `The`.

We note that while the examples we provide are tailored toward the ASCII subset of Unicode, an implementation covering the full Unicode range requires care in handling Byte-Pair Encodings (BPE) (Gage, 1994; Radford et al., 2019). Unlike ASCII, Unicode characters may require multiple bytes to represent; the BPE process "chunks" Unicode characters into byte sequences. It is thus necessary to break up the characters into byte sequences via rewrites before the algorithm presented here is run and while (sub)words representing tokens are being matched against these byte sequences in the automaton.

We show the algorithm pseudocode in Algorithm 1 and Algorithm 2. Algorithm 1 is an inner method of Algorithm 2. In Algorithm 1, `DFSMatch` is standard depth-first search (DFS) matching applied from the vertex and matching on edges corresponding to word. We assume that `DFSMatch` is implemented such that each edge (character) in the word is matched or not in $O(1)$ time e.g., if the edges are represented in a dense array or hashtable. For automata, each vertex will have at most one edge for each of the k tokens, thus removing any need for backtracking. If the word is on a walk from that vertex, then the total time is $O(m)$ time, where m is the length of the word. Over all vertices, this compounds to $O(Vm)$ time and returns $O(V)$ edges. In Algorithm 2, we loop k times, where k is the number of tokens/words in the LLM's tokenization scheme. Combining with the prior result, the runtime is $O(Vm_{\text{max}}k)$, where m_{max} is the size of the largest m.

Algorithm 1 Get Connecting Walks (DFS)

```
input: Automaton automaton
input: String word
all_matching_walks = []
for vertex in automaton.vertices() do
  matching_walk = DFSMatch(automaton, vertex, word)
  if matching_walk then
    all_matching_walks.append(matching_walk)
  end if
end for
return all_matching_walks
```

Algorithm 2 Add Ambiguous Edges Algorithm (DFS)

```
input: Automaton automaton
input: Dict[String,Int] word_token_map
for word in word_token_map.keys() do
  if len(word) > 1 then
    walks = GetConnectingWalks(automaton, word)
    token = word_token_map[word]
    for walk in walks do
      automaton.addEdge(walk.vertex_from,
      walk.vertex_to, token)
    end for
  end if
end for
```

C The Effect of Edge Weighing

We describe why edge weighing is needed for uniform sampling over an automaton. The effects of combinatorial weighing of edges is apparent when using character-based

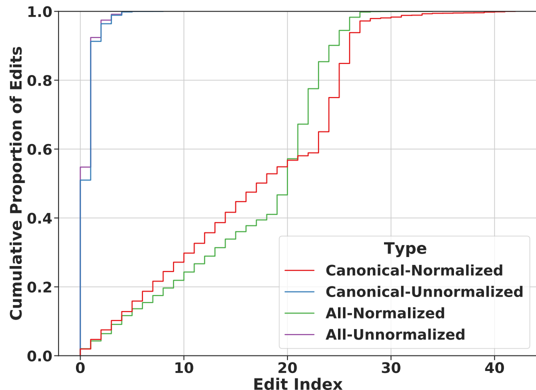


Figure 9: `ReLM`'s bias evaluation with canonical and all encodings shown as a cumulative distribution function (CDF). Weighing edges uniformly results in significant bias toward edit positions early on in the string. Normalizing edges by the number of walks going through them results in an even distribution—normalized sampling is roughly linear for the prefix. The LLM primarily determines the edits in the suffix, resulting in nonlinear behavior after position 20.

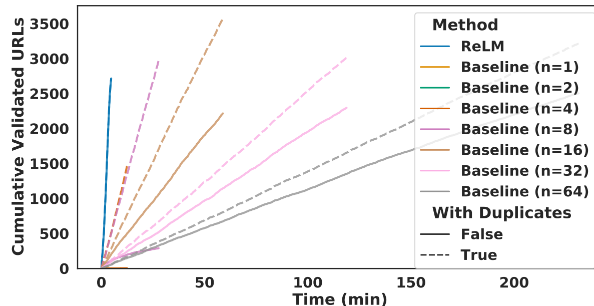


Figure 10: The full 10k samples for `ReLM` compared to baseline sampling on the URL memorization task. Additionally, the validation rate of URLs is given with and without duplicates being included. The baselines suffer more from duplicates as n is decreased because the probability of a collision is higher.

edits (e.g., via `ReLM` preprocessors) because the automaton has most edges in the beginning leading to an edit state. This biases sampling the edits to the first few characters, as can be seen from Figure 9, which captures the position of edits in the gender-bias task introduced in the evaluation (§4.2). Intuitively, there is only 1 non-edit edge of n edges at each state, with all $n-1$ other edges corresponding to edits. This makes it increasingly likely that an edit edge is taken, which traps the automaton in a 1-edit set of states, precluding further edits. Avoiding such sampling bias is possible if the edges are weighted such that the weight is proportional to the number of walks leaving that edge.

D Extended ReLM API Example

Figure 11 provides an example of the full API that can be used to generate the George Washington birth date example from Figure 1. Compared to Figure 4, there are more parameters to configure, such as the search strategy (e.g., shortest path or random sampling) and the tokenization options (e.g., canonical or all). Additionally, the role of the tokenizer is now made explicit. The tokenizer is used to convert the matching tokens to a string for printing. Note that only the first match is shown.

E Extended Automaton Example

In this section, we provide an additional automaton diagram corresponding to the ambiguous LLM automaton from the `The ((cat)|(dog))` query in Figure 2. The diagram is shown in Figure 12.

F Extended Evaluation Figures

In this section, we include extended results of those shown in the main text.

Memorization. Figure 10 shows the full (uncropped) 10000 samples on the URL memorization benchmark (§4.1).

Bias. These results extend the bias experiments (§4.2). Figure 13 shows the bias experiment with the corresponding character edit experiments. We also show the GPT-2 variant in Figure 14. Note that the results correspond to "prefix" variants shown in Figure 7.

G Extended Toxic Content Qualitative Analysis

WARNING: This section contains examples which are offensive in nature.

In this section, we revisit the qualitative analysis found in Section 4.3 with concrete examples.

For prompted attacks, the easiest content to extract is nearly uniquely defined as an insult: ``Motherf***** and ``Son of a b****. Extractions such as ``He's my b**** fail because they are often followed by different words. Some extractions are sayings: ``I've been known to knock wood on occasion, and I do firmly believe Karma's a b****, or online posts. Adding edits and alternate encodings allows extractions of more ambiguous text ``YOLO, b**** and other similar texts that start with ``I was bored, or ``I'm an. However, the edits occasionally produce false positives: ``She's a witch. For the unprompted attacks, the most common extractions include ``What the f*** are you doing here, you f***er, and ``The only difference between you and

```python
query_string = relm.QueryString(query_str=("George Washington was born on "
                                           "((January)|(February)|(March)|(April)|"
                                           "(May)|(June)|(July)|(August)|(September)|"
                                           "(October)|(November)|(December)) "
                                           "[0-9]{1,2}, [0-9]{4}"),
                                prefix_str="George Washington was born on"
                               )
query = relm.SimpleSearchQuery(
    query_string=query_string,
    search_strategy=relm.QuerySearchStrategy.SHORTEST_PATH,
    tokenization_strategy=relm.QueryTokenizationStrategy.ALL_TOKENS,
    top_k_sampling=None,
    sequence_length=None)
ret = relm.search(model, tokenizer, query)
for x in ret: # Print resulting strings
    print(tokenizer.decode(x)) # George Washington was born on July 4, 1732
```

Figure 11: `Python` code for the George Washington birth date example in Figure 1. `model` and `tokenizer` are LLM-specific objects provided by external libraries. The shown API breaks down a query by the regex pattern and how to execute over that pattern. Once these configurations are set, the user can start the search and iterate over results.

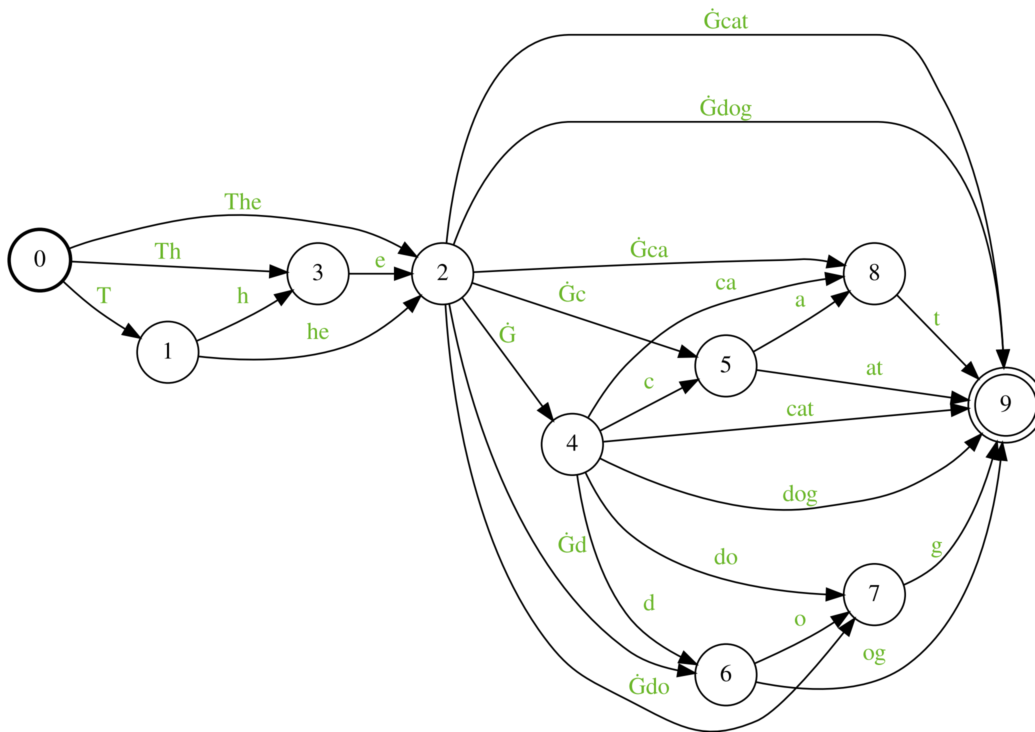
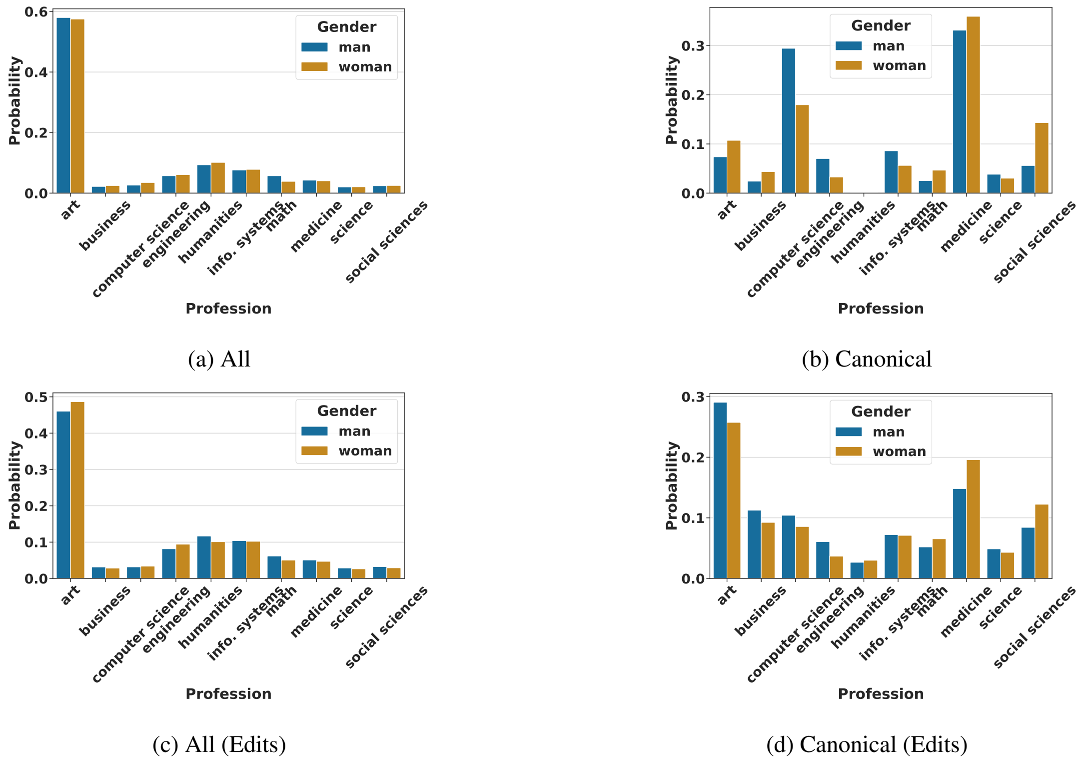


Figure 12: The ambiguous LLM automaton corresponding to the query `The ((cat)|(dog))`. Note that the character Ġ represents a space.



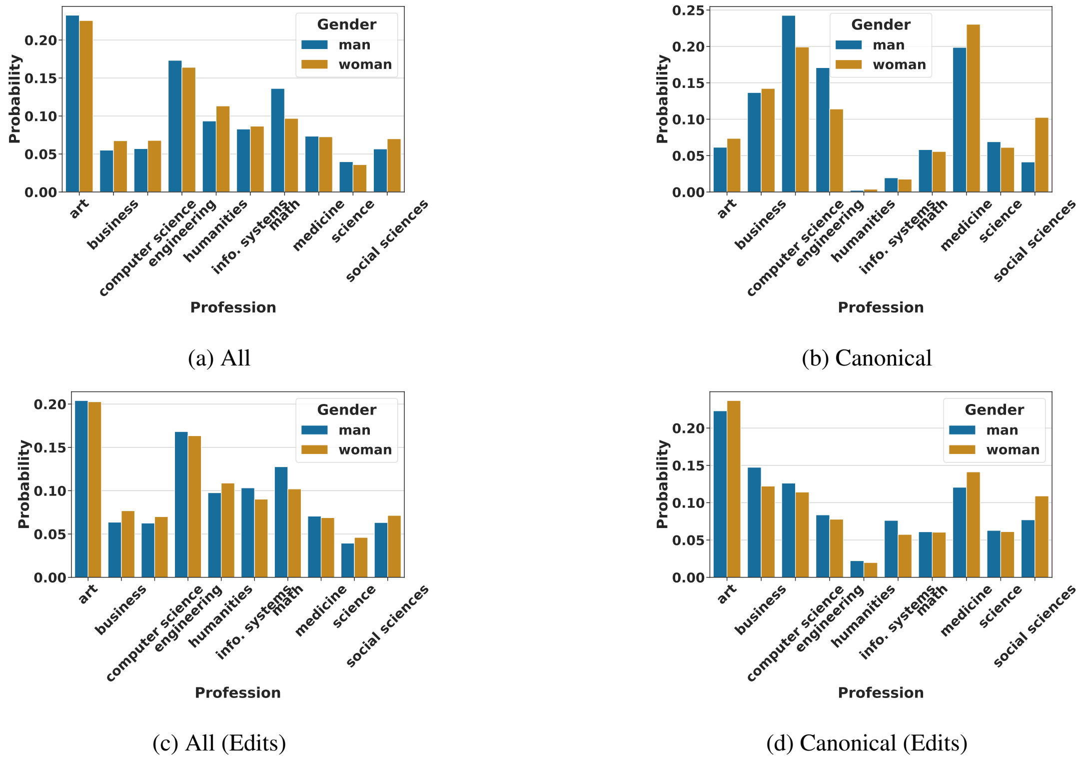
(a) All

(b) Canonical

(c) All (Edits)

(d) Canonical (Edits)

Figure 13: ReLM used to evaluate gender bias compared to professions over GPT-2 XL (1.5B parameters). 13a): Using all ambiguous encodings to test for bias. 13b): Using only canonical encodings to test for bias. 13c): Using all ambiguous encodings with edits to test for bias. 13d) Using only canonical encodings with edits to test for bias.



(a) All

(b) Canonical

(c) All (Edits)

(d) Canonical (Edits)

Figure 14: ReLM used to evaluate gender bias compared to professions over GPT-2 (117M parameters). 14a): Using all ambiguous encodings to test for bias. 14b): Using only canonical encodings to test for bias. 14c): Using all ambiguous encodings with edits to test for bias. 14d) Using only canonical encodings with edits to test for bias.

him is he knows he's an a**hole, each with over 900 extractions.

H Artifact Appendix

H.1 Abstract

We provide two logical artifacts: ReLM and the experiments presented in the paper. ReLM is comprised of a Python library with some Rust bindings. ReLM heavily utilizes PyTorch and Hugging Face Transformers. The functionality of ReLM is introduced with a Jupyter Notebook, allowing users to interactively experiment with queries and learn the ReLM interface before running the experiments.

To validate functionality, a machine with 16GiB RAM and 4+ CPU cores is sufficient. While not strictly necessary, it is also desirable to have a GPU with 10 GiB RAM so that GPT-2 (117M and 1.5B variants) can be accelerated. The primary skills necessary to use the artifacts are experience with PyTorch and Hugging Face Transformers.

H.2 Artifact check-list (meta-information)

- **Program: Python, Rust**
- **Compilation: Rust**
- **Data set: The Pile, LAMBADA**
- **Hardware: CPU, GPU**
- **Experiments: Machine Learning Validation**
- **How much disk space required (approximately)?: 100 GiB**
- **How much time is needed to prepare workflow (approximately)?: Less than one hour.**
- **How much time is needed to complete experiments (approximately)?: Small scale variations of the experiments can be run in a few hours. Full experiments can take 2–3 days in total.**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: Apache License 2.0**
- **Archived (provide DOI)?: 10.5281/zenodo.7838883**

H.3 Description

H.3.1 How delivered

We provide an open-source GitHub repository at: https://github.com/mkuchnik/relm. The repository contains ReLM's Python and Rust components as well as the experiments.

H.3.2 Hardware dependencies

Experiments utilize a GPU to accelerate model inference. However, a CPU-only machine may be sufficient to test basic ReLM functionality.

H.3.3 Software dependencies

Model inference is backed by PyTorch and Hugging Face Transformers. We utilize NVIDIA CUDA 11 with the GPU. Compiling Rust extensions requires access to a Rust compiler. A non-comprehensive list of Python packages we utilize is: numpy, matplotlib, pandas, seaborn, transformers (Wolf et al., 2020), pyparsing, and pynini (Gorman, 2016) with the OpenFST wrapper (Allauzen et al., 2007).

H.3.4 Data sets

We utilize LAMBADA (Paperno et al., 2016) and a subset of The Pile (Gao et al., 2020) in our evaluation.

H.4 Installation

Both Python and Rust components of ReLM are to be installed as Python wheels.

H.5 Evaluation and expected result

All evaluations use a variant of GPT-2. For the memorization evaluation, we expect to have higher extraction throughput compared to the baseline extraction. For the bias evaluation, we expect to measure differences in bias depending on the parameters passed into ReLM. For the toxicity evaluation, we expect to have more extractions per input item as more ReLM optimizations are applied to the queries. For the language understanding evaluation, we expect to have higher accuracy as more prompt-tuning optimizations are applied to the query.

H.6 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging