Infinigram: Corpus-Based Language Modeling via Suffix Arrays with LLM Probability Mixing

Technical Report

December 14, 2025

Abstract

We present Infinigram, a corpus-based language model that uses suffix arrays for variable-length n-gram pattern matching. Unlike neural language models that require expensive training and fine-tuning, Infinigram provides instant "training"—the corpus *is* the model. Given a context, Infinigram finds the longest matching suffix in the training corpus and estimates next-token probabilities from observed continuations. This approach offers three key advantages: (1) $O(m \log n)$ query time enabling real-time predictions, (2) complete explainability since every prediction traces to specific corpus evidence, and (3) the ability to ground large language model (LLM) outputs through probability mixing without retraining. We introduce a theoretical framework that views inductive biases as *projections*—transformations applied to queries or training data that enable generalization. Runtime transforms project queries to find better corpus matches, while corpus augmentations project training data to expand coverage. This unified perspective provides a principled approach to out-of-distribution generalization in corpus-based models. Infinigram achieves sub-10ms query latency on million-token corpora and can handle datasets exceeding 100GB through chunked memory-mapped indexing.

1 Introduction

Language modeling—predicting the next token given a context—is fundamental to modern natural language processing. The dominant paradigm involves training neural networks with billions of parameters on massive text corpora, a process that requires substantial computational resources and time. Fine-tuning these models for specific domains compounds the cost, yet is often necessary to reduce hallucinations and improve domain-specific performance.

We revisit a simpler approach: using the training corpus directly as the model. N-gram language models [Jelinek, 1991, Chen and Goodman, 1996] have a long history in NLP, but traditional implementations suffer from fixed context lengths and exponential memory growth. Variable-length Markov models [Rissanen, 1983, Begleiter et al., 2004] address the fixed-length limitation but remain computationally expensive for long contexts.

Infinigram leverages suffix arrays [Manber and Myers, 1993] to enable efficient variable-length pattern matching. Given a context, we find the longest suffix that occurs in the training corpus and use the distribution of tokens following that suffix as our prediction. This provides:

- **Instant training**: Building a suffix array is $O(n \log n)$ for a corpus of size n, with no iterative optimization.
- **Explainability**: Every prediction references specific corpus positions, enabling debugging and interpretation.

- **LLM grounding**: Neural LM outputs can be mixed with corpus-based probabilities to reduce hallucinations without retraining.

The key insight motivating Infinigram is that for many domain-specific applications, the relevant patterns *already exist* in the target corpus. Rather than hoping a neural model will learn and retain these patterns, we can retrieve them directly.

1.1 Contributions

Our contributions are:

1. A corpus-based language model using suffix arrays that provides $O(m \log n)$ query complexity for context length m and corpus size n.
2. A framework for *LLM probability mixing* that grounds neural model outputs in specific corpora without fine-tuning.
3. A theoretical perspective on inductive biases as *projections*—transformations of queries or data that enable generalization—with implementations of both runtime query transforms and corpus augmentations.
4. An open-source implementation with memory-mapped indexing supporting corpora exceeding 100GB.

2 Method

2.1 Suffix Array Foundation

A suffix array [Manber and Myers, 1993] for a text T of length n is an array SA of integers in $[0, n-1]$ that specifies the lexicographic ordering of all suffixes of T. Formally, $SA[i]$ gives the starting position of the i-th smallest suffix. Suffix arrays can be constructed in $O(n)$ time using algorithms such as SA-IS [Nong et al., 2009] or DC3 [Kärkkäinen and Sanders, 2003], though we use the practical $O(n \log n)$ divsufsort library for its speed in practice.

Given a pattern P of length m, we can find all occurrences in T using binary search on SA in $O(m \log n)$ time. This is the key operation underlying Infinigram's efficiency.

2.2 Variable-Length Suffix Matching

Traditional n-gram models use a fixed context length. Infinigram instead finds the *longest matching suffix*. Given a context $c = c_1 c_2 \ldots c_m$, we search for progressively longer suffixes starting from the full context:

```
function LongestSuffix($c$, $SA$, $T$)
    for $\ell = m$ down to 1 do
        suffix ← $c[m - \ell + 1 : m]$
        if Count(suffix, $SA$, $T$) $> 0$ then
            return positions, $\ell$
        end if
    end for
    return $\emptyset$, 0
end function
```

This can be optimized by noting that if suffix s is not found, no longer suffix containing s as a suffix can be found either. In practice, we start from the longest and stop at the first match.

2.3 Probability Estimation

Given the longest matching suffix of length ℓ occurring at positions $\{p_1, \ldots, p_k\}$, we estimate next-token probabilities from the tokens following these positions. Let $\text{count}(t)$ be the number of positions where token t follows the matched suffix:

$$P(t \mid c) = \frac{\text{count}(t) + \alpha}{\sum_{t'} \text{count}(t') + \alpha \cdot |V|} \tag{1}$$

where α is a Laplace smoothing parameter and $|V| = 256$ for byte-level modeling. This ensures non-zero probabilities for all possible continuations.

2.3.1 Hierarchical Weighted Prediction

Rather than using only the longest match, we can combine predictions from multiple suffix lengths:

$$P(t \mid c) = \frac{\sum_{\ell=1}^{L} w(\ell) \cdot \text{count}_\ell(t)}{\sum_{\ell=1}^{L} w(\ell) \cdot \text{total}_\ell} \tag{2}$$

where $w(\ell)$ is a weighting function. Infinigram supports linear ($w(\ell) = \ell$), quadratic ($w(\ell) = \ell^2$), exponential ($w(\ell) = b^\ell$), and sigmoid weighting functions. Longer matches receive higher weights since they provide more specific context.

2.3.2 Stupid Backoff

We also implement Stupid Backoff [Brants et al., 2007], which uses the longest reliable match and backs off to shorter contexts with a penalty factor λ (typically 0.4):

$$S(t \mid c) = \begin{cases} \text{count}(t \mid c)/\text{total} & \text{if count}(c) \geq \tau \\ \lambda \cdot S(t \mid c_{2:m}) & \text{otherwise} \end{cases} \tag{3}$$

This is faster than hierarchical prediction since it stops at the first sufficiently reliable match.

2.4 LLM Probability Mixing

The primary practical application of Infinigram is *grounding* neural language model outputs in specific corpora. Given an LLM with next-token distribution P_{LLM} and a corpus-specific Infinigram model with distribution P_{IG}, we compute:

$$P_{\text{final}}(t \mid c) = \alpha \cdot P_{\text{LLM}}(t \mid c) + (1 - \alpha) \cdot P_{\text{IG}}(t \mid c) \tag{4}$$

where $\alpha \in [0, 1]$ controls the mixing ratio. This provides several benefits:

- **Domain adaptation without retraining**: Mixing with a legal corpus boosts legal terminology without fine-tuning.
- **Hallucination reduction**: Corpus-based probabilities anchor outputs to text that actually exists.

- **Real-time adaptation**: Switching corpora is instant—no retraining required.
- **Interpretability**: When the model produces unexpected output, we can trace the corpus contribution.

The mixing parameter α can be fixed or adaptive based on the Infinigram model's confidence (match length and frequency). Low confidence indicates the corpus lacks relevant patterns, so the LLM should dominate.

2.5 Byte-to-Token Marginalization

A practical challenge for probability mixing is that Infinigram operates at the *byte level* (vocabulary size 256), while LLMs typically use subword tokenizers with vocabularies of 32k–100k+ tokens. Directly mixing probabilities requires compatible token spaces.

We solve this through *byte-to-token marginalization*: computing token probabilities from byte probabilities via the chain rule. For a token t that encodes to byte sequence $[b_1, b_2, \ldots, b_k]$:

$$P_{\mathrm{IG}}(t \mid c) = \prod_{i=1}^{k} P(b_i \mid c, b_1, \ldots, b_{i-1}) \tag{5}$$

This is *exact*, not an approximation. A token is simply a named byte sequence, and its probability is the joint probability of its constituent bytes under the chain rule.

Example. Consider context "Hello" and token " world" (with leading space), which encodes to bytes $[32, 119, 111, 114, 108, 100]$:

$$\begin{aligned} P_{\mathrm{IG}}(\text{`` world''} \mid \text{``Hello''}) = {} & P(32 \mid \text{``Hello''}) \\ & \times P(119 \mid \text{``Hello ''}) \\ & \times P(111 \mid \text{``Hello w''}) \\ & \times P(114 \mid \text{``Hello wo''}) \\ & \times P(108 \mid \text{``Hello wor''}) \\ & \times P(100 \mid \text{``Hello worl''}) \end{aligned} \tag{6}$$

Each factor is a single Infinigram prediction, extending the context as we generate each byte.

Efficiency considerations. Computing token probabilities for all 100k+ tokens in a typical LLM vocabulary would be prohibitive. In practice, we only compute $P_{\mathrm{IG}}(t)$ for the top-k tokens from the LLM distribution (typically $k = 50$–100), since the remaining tokens have negligible probability anyway.

For numerical stability, we compute in log space:

$$\log P_{\mathrm{IG}}(t \mid c) = \sum_{i=1}^{k} \log P(b_i \mid c, b_1, \ldots, b_{i-1}) \tag{7}$$

and use log-sum-exp for the mixture:

$$\log P_{\mathrm{final}}(t) = \log \left(\alpha \cdot e^{\log P_{\mathrm{LLM}}(t)} + (1 - \alpha) \cdot e^{\log P_{\mathrm{IG}}(t)} \right) \tag{8}$$

This approach has a key advantage: *one byte-level index serves all LLM tokenizers.* We build the suffix array once over raw bytes and can compute token probabilities for GPT-4, Claude, Llama, or any other model at query time without rebuilding the index.

3 Projections as Inductive Biases

A central theoretical contribution is viewing inductive biases as *projections*—transformations that map queries or training data to enable better generalization. This framework unifies several techniques under a common abstraction.

3.1 The Projection Framework

Let $\mathcal{Q}$ be the space of possible queries and $\mathcal{D}$ the training data space. An inductive bias helps the model generalize from training data to novel queries. We observe that many useful inductive biases can be expressed as projections:

- A **query projection** $\pi_Q : \mathcal{Q} \to \mathcal{Q}$ transforms the query before matching.
- A **data projection** $\pi_D : \mathcal{D} \to \mathcal{D}$ transforms training data, typically creating augmented variants.

The key insight is that instead of building complex models with implicit biases, we can achieve similar effects through explicit, interpretable transformations.

3.2 Runtime Query Transforms

Infinigram supports runtime query transformations that project queries to find better corpus matches:

- `lowercase`: Convert query to lowercase, enabling case-insensitive matching.
- `casefold`: Unicode case folding for language-independent case normalization.
- `strip`: Remove leading/trailing whitespace.
- `normalize_whitespace`: Collapse multiple whitespace to single spaces.

For example, the query "THE CAT" with `lowercase` transform will match corpus occurrences of "the cat". This is a projection: we map the query to a normalized form where matching is more likely.

Transforms can be composed sequentially. The model also supports *beam search* over transform combinations, exploring multiple projected queries and combining their predictions with learned weights.

3.3 Corpus Augmentation as Data Projection

Corpus augmentations apply projections to training data, creating additional variants that expand pattern coverage. Given documents $D = \{d_1, \ldots, d_k\}$ and augmentation functions $\{a_1, \ldots, a_m\}$, the augmented corpus contains:

$$D' = D \cup \bigcup_{i,j} \{a_j(d_i)\} \tag{9}$$

For example, with a lowercase augmentation, the corpus containing "Hello World" also contains "hello world". Queries can then match either variant without runtime transformation.

3.4 Duality and Trade-offs

Query transforms and corpus augmentations are dual mechanisms achieving similar effects:

Aspect	Query Transform	Corpus Augmentation
Storage cost	No increase	Multiplicative
Query cost	Transform + search	Single search
Flexibility	Runtime choice	Build-time choice
Consistency	Applied per-query	Always available

Query transforms are preferable when storage is limited or flexibility is important. Corpus augmentations are preferable when query latency is critical or certain projections should always apply.

3.5 Theoretical Perspective

This projection framework suggests a path toward more sample-efficient corpus-based models. Complex neural inductive biases—such as attention patterns, positional encodings, or architectural constraints—implicitly project inputs to representations where patterns are easier to learn. Our explicit projections make these transformations interpretable and composable.

Future work might explore richer projections: semantic clustering (map tokens to concept IDs), lemmatization (map to root forms), or embedding-based soft matching (find semantically similar contexts even without exact string overlap).

4 Implementation

Infinigram is implemented in Python with performance-critical operations delegated to C libraries.

4.1 Architecture

The system has four layers:

1. **Suffix Array Engine**: Uses `pydivsufsort` for $O(n \log n)$ construction at 15–30 MB/s. Supports memory-mapped storage for corpora larger than RAM.

2. **Core Model**: The `Infinigram` class provides `predict()`, `predict_weighted()`, and `predict_backoff()` methods with configurable smoothing and transforms.

3. **REST API**: A FastAPI server provides OpenAI-compatible endpoints (`/v1/completions`) for integration with existing tooling.

4. **REPL**: An interactive shell for exploration, dataset management, and ad-hoc queries.

4.2 Byte-Level Tokenization

Infinigram operates on raw UTF-8 bytes, giving a fixed vocabulary of 256 tokens. This avoids tokenizer dependencies and vocabulary mismatch issues when mixing with different LLMs. The trade-off is longer effective context lengths (characters require 1–4 bytes), mitigated by the efficiency of suffix array queries.

4.3 Memory-Mapped Indexing

For large corpora, Infinigram uses memory-mapped files for both the corpus and suffix array. This allows the operating system to page in only the needed portions, enabling queries on 100GB+ corpora with minimal RAM usage. For corpora exceeding available memory during construction, a chunked index splits the data into independently indexed segments whose results are merged at query time.

4.4 Performance

Target performance characteristics:

- **Construction**: 1M tokens/second (15–30 MB/s for byte-level)
- **Query latency**: <10ms for 100-token contexts
- **Memory**: <10 bytes per corpus token (8 bytes for suffix array + corpus)

5 Related Work

N-gram Language Models Traditional n-gram models [Jelinek, 1991, Chen and Goodman, 1996] estimate $P(w_n \mid w_1, \dots, w_{n-1})$ from corpus counts with various smoothing techniques (Kneser-Ney, Witten-Bell). These models use fixed context lengths, limiting their ability to capture long-range dependencies. Infinigram's variable-length matching addresses this limitation.

Suffix Arrays and Trees Suffix arrays [Manber and Myers, 1993] and suffix trees [Weiner, 1973, McCreight, 1976] enable efficient string matching. The Burrows-Wheeler Transform and FM-index [Ferragina and Manzini, 2000, 2005] provide compressed alternatives. Our use of suffix arrays for language modeling builds on this algorithmic foundation.

Retrieval-Augmented Generation RAG systems [Lewis et al., 2020, Izacard et al., 2022] retrieve relevant documents to condition neural generation. Infinigram differs by retrieving *exact pattern matches* rather than semantically similar documents, and by modifying token-level probabilities rather than prepending context. The approaches are complementary.

kNN-LM Khandelwal et al. [2020] propose augmenting neural LMs with a nearest-neighbor lookup in a datastore of context embeddings. This requires storing embeddings for every training token (hundreds of GB for large corpora) and uses approximate nearest neighbor search. Infinigram uses exact string matching with much lower storage overhead.

Infini-gram Concurrent work by Liu et al. [2024] presents a system also named "infini-gram" that indexes massive web corpora (5 trillion tokens) for n-gram analysis and LLM augmentation. Their focus is on scaling to web-scale data, while our work emphasizes the projection framework for generalization and practical domain-specific applications.

6 Conclusion and Future Work

Infinigram provides a simple, fast, and explainable approach to language modeling that complements neural methods. The ability to mix corpus-based probabilities with LLM outputs offers a practical alternative to fine-tuning for domain adaptation. Our projection framework provides a theoretical lens for understanding generalization in corpus-based models.

Future directions include:

- **Richer projections**: Semantic clustering, lemmatization, and edit-distance-based fuzzy matching could expand coverage without proportional corpus growth.
- **Adaptive mixing**: Learning context-dependent α values for LLM probability mixing based on match confidence and domain signals.
- **Compressed indexes**: FM-index or grammar-compressed representations could reduce storage for highly repetitive corpora.
- **Multi-scale models**: Combining byte-level, subword, and word-level indexes to balance specificity and coverage.

The source code is available at `https://github.com/example/infinigram` under an open-source license.

References

Ron Begleiter, Ran El-Yaniv, and Golan Yona. On prediction using variable order Markov models. *Journal of Artificial Intelligence Research*, 22:385–421, 2004.

Thorsten Brants, Ashok C Popat, Peng Xu, Franz J Och, and Jeffrey Dean. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 858–867, 2007.

Stanley F Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 310–318, 1996.

Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 390–398, 2000.

Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4): 552–581, 2005.

Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. Atlas: Few-shot learning with retrieval augmented language models. *arXiv preprint arXiv:2208.03299*, 2022.

Frederick Jelinek. Up from trigrams! the struggle for improved language models. *Proceedings of Eurospeech*, pages 1037–1040, 1991.

Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, pages 943–955, 2003.

Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through memorization: Nearest neighbor language models. In *International Conference on Learning Representations*, 2020.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474, 2020.

Jiacheng Liu, Sewon Min, Luke Zettlemoyer, Yejin Choi, and Hannaneh Hajishirzi. Infini-gram: Scaling unbounded n-gram language models to a trillion tokens. *arXiv preprint arXiv:2401.17377*, 2024.

Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

Edward M McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. In *IEEE International Symposium on Information Theory*, pages 449–453, 2009.

Jorma Rissanen. A universal data compression system. *IEEE Transactions on Information Theory*, 29(5):656–664, 1983.

Peter Weiner. Linear pattern matching algorithms. *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.