Intermediate N-Gramming: Deterministic and Fast N-Grams For Large N and Large Datasets

Ryan R. Curtin[1], Fred Lu[1,2], Edward Raff[2,3], Priyanka Ranade[4]

[1]Booz Allen Hamilton
[2]University of Maryland, Baltimore County
[3]CrowdStrike
[4]Laboratory for Physical Sciences

Abstract

The number of n-gram features grows exponentially in n, making it computationally demanding to compute the most frequent n-grams even for n as small as 3. Motivated by our production machine learning system built on n-gram features, we ask: is it possible to accurately, deterministically, and quickly recover the top-k most frequent n-grams? We devise a multi-pass algorithm called *Intergrams* that constructs candidate n-grams from the preceding $(n-1)$-grams. By designing this algorithm with hardware in mind, our approach yields more than an order of magnitude speedup (up to 33×!) over the next known fastest algorithm, even when similar optimization are applied to the other algorithm. Using the empirical power-law distribution over n-grams, we also provide theory to inform the efficacy of our multi-pass approach.

Code — https://github.com/rcurtin/Intergrams
Extended version — https://arxiv.org/abs/2511.14955

Introduction

The goal of this work is the problem statement: "*I should be able to find the top-k n-grams of a dataset as quickly as the disk can give me bytes*". If one has a large set of files (TBs or larger), and wants to find the k most frequent subsequences of length n in that set of files (called *n-grams*), the claim is that the bottleneck in computation should be getting the bytes from the files—*not* processing them. This problem is *much* harder than it looks from the description, but it *is* possible to satisfy the claim.

The problem is of interest not because it was lunchtime banter (although it was), but instead because it is an important preprocessing step for numerous real-world machine learning tasks: n-grams are effective features as inputs to machine learning models. Once the top-k n-grams are known, it is extremely fast to convert a large set of files into sparse data that can then be directly used for training—see Fig. 1. But, for large datasets, k, or n, the time to compute the top-k n-grams can significantly exceed all other steps.

Probably the most well-known application of this strategy, is malware classification: byte-sequence features are commonly used to produce simple, interpretable models that give state-of-the-art performance and extremely fast inference times (Masud, Khan, and Thuraisingham 2008; Fuyong and Tiezhu 2017; Raff and Nicholas 2018). For very large datasets, they may even be used for distributed training of models (Lu et al. 2025, 2024). For text processing, length-n character sequences have been shown to outperform text embedding models for (certain) textual event classification (Piskorski and Jacquet 2020) and are useful in large-scale language modeling (Liu et al. 2024). Finally, n-grams of genomic sequences (known as k-mers) have proven to be useful for various tasks, from phylogenetic analysis to species classification (Chor et al. 2009; Yang et al. 2020).

Copyright © 2026, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

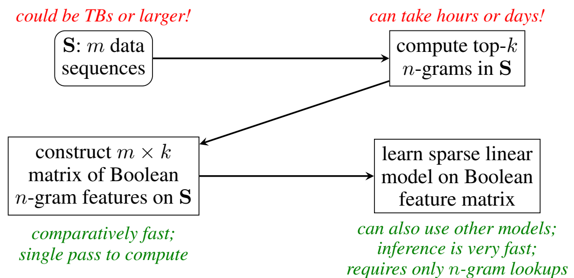

Figure 1: A typical machine learning pipeline using n-grams as features. To run this pipeline, the k most common n-grams from the input data must be known. Often, on large datasets, the process of computing the top-k n-grams is far more expensive than the modeling step!

Problem Formalization and Notation

The top-k n-gram problem is straightforward: suppose there is a dictionary $\mathbf{D}$ of possible n-grams; it may be extremely large. If n-grams are byte sequences, then $|\mathbf{D}| = 256^n$ (e.g. there are 18 quintillion possible 8-grams!). Suppose also there is a dataset that consists of m sequences $\mathbf{S} = \{s_1, s_2, \ldots, s_m\}$. In a malware detection application, for instance, $\mathbf{S}$ may be a collection of files where each s_i is one file. Our task is to find one of the following two quantities:

$$k\text{-argmax}_{x \in \mathbf{D}} \sum_{s_i \in \mathbf{S}} \sum_{x_j \in s_i} \mathbb{1}(x_j = x), \tag{1}$$

$$k\text{-argmax}_{x \in \mathbf{D}} \sum_{s_i \in \mathbf{S}} \mathbb{1}(x \in s_i). \tag{2}$$

In the first variant (Eq. 1), each n-gram is counted once *every* time that it appears. In the second variant (Eq. 2), an n-gram is only counted once for each sequence s_i that it appears in. When using the top-k n-grams as features for a machine learning model, especially on real-world data, it is generally more effective to count an n-gram once per sequence. This is because it is often the case that a single n-gram may be present in only a very few sequences of $\mathbf{S}$, but may occur very many times in those few sequences. If our goal is to predict the label of a sequence, then these n-grams carry little predictive value, as their feature values will be 0 for the vast majority of sequences in $\mathbf{S}$. For more details on this phenomenon, see the discussion in Raff et al. (2018).

As a result, our focus in this paper will be on solving Eq. 2; however, our algorithms can be readily adapted to solve Eq. 1 (and they will solve that problem more quickly as it is an easier problem).

Related Work

The naive approach to solving this problem consists of storing a large dictionary for each element in $\mathbf{D}$ and iterating over each sequence s_i, incrementing the appropriate counts in the dictionary. Then, when all sequences have been iterated over, sort the dictionary by count and keep the top k.

Such a strategy can be implemented in only a few lines of code; in fact, this is what is done by the commonly-used `CountVectorizer` class from the `scikit-learn` package (Pedregosa et al. 2011). But this obviously scales horrendously for non-trivial sizes of $\mathbf{D}$ or k.

A number of stream-based approaches have been developed, e.g. Charikar, Chen, and Farach-Colton (2002), Cormode and Muthukrishnan (2005), and Jin et al. (2003). These approaches build a 'sketch' of the data as the stream is processed, discarding n-grams that are found to be infrequent. Although these approaches perform better than the naive strategy described above, they tend to focus on much smaller k than would be interesting for a machine learning use case, and require storing significantly more than k partial counts. For instance, the Space-Saving algorithm of Metwally, Agrawal, and El Abbadi (2005) requires a number of counters $M >> k$ for error bounds to be acceptably small. Further, adapting these algorithms to count an n-gram only once per sequence (Eq. 2) adds additional overhead.

A more principled approach can be developed by observing that the n-gram distribution of real-world data is *not* uniform but instead follows a Zipfian (power-law) distribution (Raff et al. 2018). That is, few n-grams appear regularly; the vast majority of n-grams either appear rarely or not at all (and as such have no chance of being in the top k).

Using this reality, Raff and Nicholas (2018) proposed the 'hash-gramming' approach, which is the current best-known algorithm for finding top-k n-grams for non-trivial k. This approach computes the hash of each n-gram, mapping it from a space of size $|\mathbf{D}|$ to a more manageable number of elements; in their implementation, they hash to roughly 2^{31} elements. Processing is done in two passes: in the first pass, the hashes of each n-gram are counted. Then, the top-k hashes are computed. In the second pass, exact n-gram values using an approach like the naive dictionary approach, but *only* for n-grams whose hashes are in the set of top-k hashes. This reduces the size of memory needed for the second pass to roughly $O(k)$ memory, which is far faster and more manageable than $O(|\mathbf{D}|)$ memory! With a good implementation, the hash-gramming approach can yield throughput of $10-20$ MB/s; although this is orders of magnitude faster than both the naive dictionary algorithm and the Space-Saving algorithm, it is still far away from the 'disk speed' postulated in the introduction (which would be more like 600 MB/s for a typical desktop computer with a spinning-disk drive).

Hierarchy level	Capacity	Peak Bandwidth	Latency
L1 cache	16 KB–64 KB	1 TB+/s	~ 1 ns
L2 cache	64 KB–512 KB	1 TB+/s	~ 4 ns
L3 cache	8 MB–256 MB	100-500 GB/s	~ 40 ns
RAM	1 GB–4 TB	10-100 GB/s	~ 80 ns
Disk	1 TB+	500 MB/s–5 GB/s	$\sim 8-80\ \mu$s

Table 1: Typical performance characteristics of each level of the memory hierarchy on modern computers (Brett 2016). Whenever data is too large to fit in a cache level, access times become significantly longer, and bandwidth decreases, both by orders of magnitude.

Hardware Limitations

Although the algorithms described in the previous section focus on minimizing overall memory usage they do not focus on actual performance limitations of physical hardware. The biggest problem that all of the algorithms described in the previous section face is **non-contiguous memory access patterns** into **large blocks of memory**.

On a modern computer, data structures that cannot fit into the processor caches (L1/L2/L3) are instead stored in RAM. Each level of this memory hierarchy has orders-of-magnitude different capacity, bandwidth, and latency. Table 1 shows typical figures on modern hardware as provided by Intel (Brett 2016), including for disk speed.

An important reality somewhat masked by Table 1 is that peak bandwidth numbers are for sequential accesses; random non-sequential access into a block of memory can be orders of magnitude slower. Thus, for some array `a`, a loop that modifies `a[0]`, then `a[1]`, then `a[2]`, and so forth is much faster than a loop that modifies random elements of `a`. See Figure 2 for a simulation that shows this effect (imitating the excellent work of Drepper (2007)).

Another important reality of hardware is paging and virtual memory: when we access a random location in memory, the operating system must first map the given memory pointer to the true underlying hardware location using the virtual memory page tables. Virtual memory is organized into pages (typically of size 4 KB but this can vary) and therefore the first step in any virtual memory access is to get the underlying hardware location of the page. The processor's translation lookaside buffer (TLB) assists with this process by caching entries of recently-used pages. But the TLB's capacity is limited; typical modern TLBs may have only hundreds of entries. Whenever a page is not found in the TLB, the kernel must perform a time-consuming page walk; this is seen in Fig. 2.

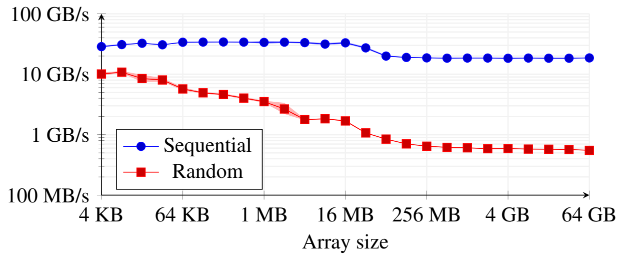

Figure 2: Memory bandwidth benchmarks. We vary the size of the array (x axis), and in each trial we increment all elements of the array in sequential (blue) or random (red) order. For sequential access patterns, the hardware prefetcher can keep bandwidth high, but the 2048-element TLB necessarily starts to miss at 8 MB (we use 4 KB page sizes). For random access patterns, the hardware prefetcher cannot predict the memory (or page) that is needed next, and throughput craters as the array size exceeds the size of each level in the cache hierarchy.

In order to design a fast algorithm for n-gramming, we must be aware of these hardware concerns. Similar to the 'cache-oblivious' algorithms pioneered by Frigo et al. (1999) and Demaine (2002), we will use a simple model of a cache and then ensure that our algorithm operates in a way that minimizes accesses outside the cache.

Hash-gramming Memory Problems

Existing algorithms for fast top-k n-gram computation severely violate the understanding of the memory hierarchy we established above. The working sets of these algorithms often significantly exceed cache size, and the memory access patterns are effectively random. Although the analysis is similar for the Space-Saving algorithm and others, we focus here on Hash-Gramming, given in Algorithm 1, as it is the fastest competitor to our algorithm by orders of magnitude and its shortcomings served heavily as inspiration.

Hash-Gramming operates by taking two passes on the data: in the first pass, a hash function is used to map an n-gram into one of B buckets. Then, the top-k buckets are retained, and in the second pass, any n-grams whose hashes are in the top-k buckets are counted; finally, the top-k n-grams of the second pass are returned. Because of the condition that an n-gram hashes to the top-k buckets, and $k \ll B$, the second pass is significantly faster.

But let us consider the memory access behavior of this algorithm: at each step in the first pass, we compute the hash of an n-gram, $h(x_i)$. $h(\cdot)$ must hash n-grams randomly to one of B buckets, but there is no correlation between the buckets that two consecutive n-grams will hash to.

Thus, each update to T_i is effectively random. Problematically, B is large: typically around 2^{31}. This means that if T_i is represented as a bit vector, its size is $\sim$8 MB. Although this fits into most L3 caches in the single-threaded case, with many threads operating on different sequences, the sum total of all bit vectors T_i can quickly exceed L3 cache.

Worse, the update step for T also exhibits random access behavior: because of the Zipfian distribution generally seen

Algorithm 1: Hash-Gramming Raff et al. (2019), specialized to count an n-gram once per sequence.

Require: Bucket size B, hash function $h(\cdot)$, sequence set $\mathbf{S}$, n, k

1: $T \leftarrow \mathbf{0}^B$ ▷ First pass: compute n-gram hash counts.
2: **for all** sequence s_i in $\mathbf{S}$ **in parallel do**
3: $\quad T_i \leftarrow \mathbf{0}^B$
4: $\quad$**for all** n-gram $x_i \in S$ **do**
5: $\quad\quad T_i[h(x_i)] \leftarrow 1$
6: $\quad$**end for**
7: $\quad T \leftarrow T + T_i$
8: **end for**
9: $H \leftarrow k\text{-argmax}_{h_i \in T}\, T[h_i]$ ▷ Top-k hash values.
10: $D \leftarrow$ empty dictionary ▷ Second pass: compute exact counts of n-grams.
11: **for all** Sequence s_i in $\mathbf{S}$ **in parallel do**
12: $\quad D_i \leftarrow$ empty dictionary
13: $\quad$**for all** n-gram $x_i \in s_i$ **do**
14: $\quad\quad$**if** $h(x_i) \in H$ **then**
15: $\quad\quad\quad D_i[x_i] \leftarrow 1$
16: $\quad\quad$**end if**
17: $\quad$**end for**
18: $\quad D \leftarrow D + D_i$
19: **end for**
return $k\text{-argmax}_{x_i \in D}\, D[x_i]$ ▷ Return final top-k n-grams.

in real-world data, each T_i is highly sparse (as most n-grams never appear). The throughput for this update step is very low, because the size of T is typically 8 or 16 GB, depending on the precision used. Even when T_i is scanned sequentially, the sparse updates to T cannot benefit from hardware prefetching and often suffer from page faults.

The second pass also exhibits the same random memory access patterns, but because the size of D is so much smaller, and because many n-grams are skipped since they are not in H, the observed runtime effect is comparatively less.

Although there are a number of small tricks that can be used to hide the latency and low bandwidth associated with the hash-gramming approach, these do not change the fundamental underlying access patterns. The use of a small LRU cache or Cuckoo hash table (Pagh and Rodler 2004) to cache common recently-seen hash values $h(x_i)$ and avoid lookups in or updates to T_i or flushes to T can be helpful, but again the Zipfian distribution of data hurts: due to the long tail of the distribution, the vast majority of n-grams are only seen once or twice and will not remain in the small cache, requiring accesses to T_i and T. Software prefetching with a circular buffer presents another alluring acceleration, but on many processors, a TLB miss on a prefetch will cause an immediate stall and page walk (Intel Corporation 2023). Given the large size of T, these TLB misses are almost guaranteed with a page size of 4 KB (the default). It is possible to use large pages of 2 MB on typical Linux systems[1], but while this pro-

[1]Linux kernel support for 1 GB pages is available, but not everywhere, and is tedious to reliably build into a distributable software product for a variety of non-user-friendly reasons, so we do not consider it in our work. As with 2 MB pages, it would provide

Algorithm 2: A fast algorithm to compute the top-k 3-grams on byte sequences, counting an n-gram once per sequence.

Require: sequence set $\mathbf{S}$, k

1: $C \leftarrow \mathbf{0}^{16777216}$ ▷ Size: 64 MB.
2: **for all** sequence s_i in $\mathbf{S}$ **in parallel do**
3: $\quad C_i \leftarrow \mathbf{0}^{16777216}$ ▷ Bit vector of size 2 MB.
4: $\quad$ **for all** 3-gram $x \in S_i$ **do**
5: $\quad\quad C_i[x] \leftarrow 1$ ▷ Random access into bit vector.
6: $\quad$ **end for**
7: $\quad C \leftarrow C + C_i$ ▷ Sequential flush.
8: **end for**
9: **return** k-$\text{argmax}_{x_i \in C}\, C[x_i]$ ▷ Compute top-k 3-grams.

vides a speedup by reducing the number of TLB misses, the size of the TLB for 2 MB pages is generally smaller, and it does not address the underlying issue.

Warmup: Fast 3-grams

To achieve fast speeds for n-gramming, we *must* address the memory access patterns and avoid random accesses into large data structures that cannot fit in processor caches and spill across many memory pages. We have two desiderata:

1. If we must do random access into a block of memory, that block of memory should fit in the processor cache.
2. If a block of memory must be larger than the processor cache, then we must access it sequentially.

When we are passing over all the sequences in $\mathbf{S}$, we will necessarily be visiting n-grams in effectively random order. Thus, when we take a pass over the data, we cannot expect to increment counts or any intermediate data structure in a sequential manner. To satisfy our first requirement, then, *our intermediate data structure must fit in the processor cache.*

Consider what would happen if we were only interested in computing the top-k 3-grams (e.g. $n = 3$). If we are using byte-sequence data, the number of possible 3-grams (that is, $|\mathbf{D}|$) is $256^3 = 16777216$. Then, for each sequence s_i, we can hold a bit vector with one bit for each of the 16M possible 3-grams, which has overall size 2 MB: this trivially fits into the processor cache. Since we cannot assume any ordering of the n-grams in the sequences s_i, our memory access pattern into this bit vector is effectively random.

After processing the sequence s_i, we can flush this bit vector to a global array of 16M elements using a sequential access pattern. When using 32-bit integers, this has size 64 MB, which still fits comfortably in cache (L3). See Algorithm 2 for a complete description of this algorithm.

This algorithm is embarrassingly parallel; each sequence can be handled by a different processor, with the only additional memory requirement that each processor needs its own 2 MB bit vector. For most processors, the sum total of memory required still fits easily in cache. On modern commodity hardware, this algorithm can compute n-grams as quickly as input sequences can be provided from disk.

a small speedup but not address the underlying issue.

There are additional optimizations we can make. First, we can collect bit vectors C_i to flush simultaneously: instead of flushing a C_i as soon as a sequence is complete, we wait until a specific number of bit vectors are ready, and then flush them all simultaneously: $C \leftarrow C + C_i + C_{i+1} + \ldots + C_{i+8}$. Second, we can accelerate each individual flush via the use of SIMD instructions. Processors that support AVX2 are able to simultaneously add 8 integers in a single instruction. In our tuned implementation, we are able to increment 8 elements of C for a single bit vector C_i with only 5 SIMD instructions. On the powerful system we use for our experiments, this achieves peak disk bandwidth ($\sim$5 GB/s!).

Now, what utility do 3-grams have when we are interested in general n-grams for $n > 3$? Intuitively, 3-grams carry information about larger n-grams: the count of a single 3-gram in $\mathbf{S}$ is the *sum* of counts for all n-grams with that 3-gram as the prefix. So, for instance, if a 3-gram x_i is in the top-k 3-grams, then it is intuitively likely that some 4-gram whose prefix is x_i will **also** be a top-k 4-gram.

Empirically validating our intuition is easy: on a dataset of 18GB of executable programs, we compute true 3-grams and true 4-grams. For a few values of k, we plot the percentage of 4-grams that are in the top-zk 3-grams, where z is a constant 'oversampling' factor. The results are shown in Fig. 3. The figure's implication is clear: if we compute the top-zk n-grams, then we can compute the top-k $(n+1)$-grams using only the top-zk n-grams as prefixes, which significantly reduces the size of the n-grams that must be counted.

Intergrams: Intermediate n-gramming

We can use the 3-gram algorithm of the previous section as a starting point for a fast n-gramming algorithm that we call **Intergrams**, following the naming of the programming language INTERCAL (Woods and Lyon 1973). The strategy is simple: suppose that we first computed the top-zk 3-grams. Then, we would compute the counts of all 4-grams whose 3-gram prefixes were in the top-zk 3-grams. Following this, we would compute the top-zk 4-grams and repeat the process for 5-grams, 6-grams, etc. This turns out to be a very effective strategy that meets our two hardware-based desiderata, and the strategy is formalized in Algorithm 3.

Importantly, at each stage of the algorithm we keep only

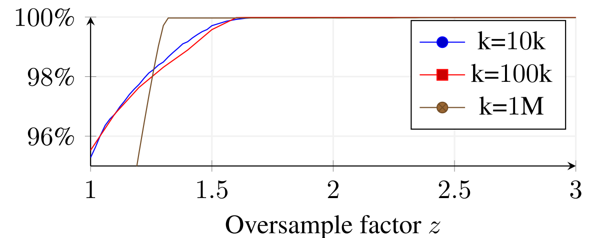

Figure 3: Percentage of top-k 4-grams that have 3-byte prefixes in the top-zk 3-grams on the EMBER dataset, as a function of z. 100% means that *all* 4-grams have 3-byte prefixes in the top-zk 3-grams; in this case, we can filter the set of possible 4-grams significantly, with no loss of accuracy!

Algorithm 3: **Intergrams**: a fast algorithm to compute the top-k n-grams on byte sequences, using $c_{\mathbb{1}}(\cdot,\cdot)$ (e.g. counting an n-gram once per sequence only).

Require: sequence set $\mathbf{S}$, n, k

1: $P^{(3)} \leftarrow$ top$-zk$ 3$-$grams using Algorithm 2
2: **for all** $j \in [4, n]$ **do**
3: $\quad C \leftarrow \mathbf{0}^{256zk}$
4: $\quad$ **for all** sequence S_i in $\mathbf{S}$ **in parallel do**
5: $\quad\quad C_i \leftarrow \mathbf{0}^{256zk}$
6: $\quad\quad$ **for all** j-gram $x \in S_i$ **do**
7: $\quad\quad\quad$ **if** x has $(j-1)$-gram prefix in $P^{(j-1)}$ **then**
8: $\quad\quad\quad\quad C_i[x] \leftarrow 1$ ▷ Random access into bit vector.
9: $\quad\quad\quad$ **end if**
10: $\quad\quad$ **end for**
11: $\quad\quad C \leftarrow C + C_i$ ▷ Sequential flush.
12: $\quad$ **end for**
13: $\quad$ **if** $j < n$ **then**
14: $\quad\quad P^{(j)} \leftarrow zk$-argmax$_{x \in C}\, C[x]$ ▷ Compute prefixes for the next round.
15: $\quad$ **end if**
16: **end for**
17: **return** k-argmax$_{x \in C}\, C[x]$

zk prefixes; this means in the next pass we only count $256zk$ n-grams. The size of each bit vector C_i will now be $32zk$ bytes; this still trivially fits in most processor caches for even large values of k and z, and thus the random access pattern into C_i is not a problem on subsequent passes.

When zk is less than 65536, the memory required for the next iteration's counts array C is less than the 64 MB needed for 3-gram computation, meaning that it too comfortably fits into the processor cache. On some systems, zk can be significantly larger before overflowing the cache. Even if C does not fit fully into the cache, the sequential access pattern of our flushes will still ensure high memory bandwidth.

It is worth considering the operation of determining whether a prefix is in P. Of course, simple iteration over all prefixes in P is very inefficient; similarly, we found that even binary search over a pre-sorted P was too slow. Instead, a trie structure (Fredkin 1960) should be used for fast lookup of a prefix's membership in P. Tries are best described visually: Fig. 4 gives an example on byte sequence data.

As lookups in this trie are the innermost computation of our entire algorithm, performance is paramount, and for this we must again consider the hardware. But we cannot make lookups sequential in a trie; we will be jumping from node to node depending on the prefix we are looking up. So our memory access pattern will appear more random than sequential. Thus, keeping the size of the trie data structure small is very important; if it is too large, it will not fit in processor cache or it will force parts of C or C_i to be evicted from cache. For byte sequence data, each trie node can have a maximum of 256 children; so we can use an 8-bit unsigned integer to track the number of children of a node. Each node must also hold a value; an 8-bit unsigned integer suffices.

During the lookup process, we must find the correct child of the trie to visit (or if there is no child to visit next, then

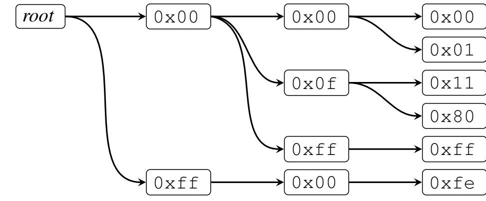

Figure 4: An example trie built on length-3 byte sequence data. Determining whether a prefix x is in P can be done by iterating from the root of the trie to a leaf, considering each element of the prefix in succession. In this trie, we can determine that the byte sequence `0x013300` is not in P at the first iteration. Other sequences, like `0x000aaf` and `0xff00ff` require more iterations; sequences like `0x000000` are in P and will reach a leaf during iteration of the trie.

we terminate early). To prevent a potentially long iteration over children when a node has many children, we can use a lookup table. For byte data, this table needs only 256 elements. We found lookup tables to be faster despite their increased memory usage for nodes with as few as 4 children.

Lastly, to reduce the number of 'faraway' memory accesses, the memory layout of the trie's children can be ordered by frequency. For example, if the prefix `0xBEEF00` is the most common prefix in $\mathbf{S}$, then the trie nodes corresponding to `0xBE`, `0xBEEF`, and `0xBEEF00` can be laid out contiguously. The second-most common child of `0xBEEF` can then be placed directly after `0xBEEF00`, and so forth. This means that the most commonly seen prefixes are more likely to have their trie elements already in the cache, and less likely to produce cache misses and page faults.

In our implementation, we were able to restrict the size of the trie to only 2 bytes per trie node plus the memory required for holding the locations of each trie's children (up to 12 bytes if the node has less than four children, and 1 KB for the lookup table otherwise; most nodes do not have lookup tables). The number of nodes in a trie is linear in the number of prefixes, so with only zk prefixes, the overall size of the trie is small compared to C. With these optimizations, overall Intergrams runtime was reduced by about 10–15%.

Theory: Intermediate Filter Recall

Next we aim to validate our claim that the top-zk n-grams can effectively filter for the top-k $(n+1)$-grams. For this analysis, we assume that n-grams and $(n+1)$-grams over $\mathcal{S}$ follow a Zipfian distribution with parameter a. That is, if the n-grams $x_1, \ldots, x_{|\mathbf{D}|}$ are ordered by decreasing probability $p_1, \ldots, p_{|\mathbf{D}|}$, then $p_i \propto \frac{1}{i^a}$. This assumption is actually pessimistic; in general, empirical data is such that the parameter a is greater for $(n+1)$-grams than for n-grams.

Although Intergrams does not explicitly search for the exact top-k $(n+1)$-grams over $\mathcal{S}$, we are able to show that with high probability, most of the probability mass of the true top-k $(n+1)$-grams is retained. To show this we first consider the ideal case where the observed n-gram counts exactly match the underlying Zipfian frequencies (e.g., no sampling noise). We will use the following facts about the Zipf distribution.

Observation 1. *Let $M_k := \sum_{i=1}^{k} \frac{1}{i^a}$, and let $|\mathbf{D}|$ be the maximum number of distinct n-grams.*

1. *The probability of sampling n-gram x_i is $p_i = \frac{i^{-a}}{M_{|\mathbf{D}|}}$.*
2. *The probability of an independently sampled n-gram being in the top-k most likely n-grams is $M_k/M_{|\mathbf{D}|}$.*

We observe that for each sequence $s_i \in \mathbf{S}$, each n-gram (except for the final one) is the prefix of an $(n+1)$-gram. Given $|\mathbf{S}| = m$ such sequences and N collected n-grams across $\mathbf{S}$, then these are the prefixes of $N - m$ corresponding $(n+1)$-grams. By this reasoning, if the top-zk n-grams account for some proportion β of all n-grams, then the proportion of $(n+1)$-grams prefixed by a top-zk n-gram is roughly β as well:

Lemma 1. *Suppose the top-k' n-grams account for proportion β of the total n-grams over dataset $\mathbf{S}$. Then the $(n+1)$-grams prefixed by one of these top-k' n-grams account for at least $\beta' := \beta - \frac{m}{N-m}$ of the total $(n+1)$-grams, with N the total n-grams over $\mathbf{S}$ and m the number of sequences.*

In the worst case scenario for Intergrams, the top-zk n-grams prefix only trivial $(n+1)$-grams, dispersing the mass β among less frequent $(n+1)$-grams. Yet, if β is large enough, then even in this adversarial situation we may still collect most frequent $(n+1)$-grams. Let X_k^n be the set of $(n+1)$-grams prefixed by a top-zk n-gram. Our goal is to lower bound the occurrence of $(n+1)$-grams within X_k^n that are also among the top-k $(n+1)$-grams.

Specifically, let u be the index of the most frequent $(n+1)$-gram within X_k^n. Then the probability mass between x_u and x_k, the k-th most frequent $(n+1)$-gram, lower bounds the relevant mass of $(n+1)$-grams that Intergrams will retain in the intermediate round.

Theorem 1. *Suppose that n-grams and $(n+1)$-grams exactly follow the Zipf distribution (no sampling noise) with parameter $a \neq 1$.[2] If Intergrams keeps the top k' n-grams from the previous pass, then (1) the most frequent $(n+1)$-gram it finds will be at least the u-th most frequent actual $(n+1)$-gram, where*

$$u \leq \Big(\big(|\mathbf{D}_{n+1}|^{1-a} - a \big)(1 - \beta') + 1 \Big)^{\frac{1}{1-a}} - 1. \quad (3)$$

(2) Furthermore, the j-th most frequent $(n+1)$-gram found will be at least the $(u+j-1)$-th most frequent.

(3) Finally, the fraction of top-k $(n+1)$-grams recalled by Intergrams is at least

$$1 - \frac{(|\mathbf{D}_{n+1}|^{1-a} - a)(1-\beta') - a}{(k+1)^{1-a} - 1}. \quad (4)$$

Intuitively, the faster the tail of the distribution decays, the higher proportion the top-k represents. When a is not too tiny, the top-k n-grams can represent a majority of observed counts, which forces the majority of $(n+1)$-grams to come from these prefixes. For larger a, such as $a > 1$, the bound Eq. 4 can be algebraically inverted to yield asymptotic behavior of the form $1 - \mathcal{O}((k/|D_{n+1}|)^{a-1})$, which quickly converges toward successful results for Intergrams.

[2]This can be re-derived for $a = 1$ and is conceptually the same.

Now, consider the case where sampling error is present: henceforth, the counts over $s_i \sim \mathcal{S}$ follow a multinomial generation model with N observations and parameters $\{p_i\}_{i=1}^{|\mathbf{D}|}$, with empirical probabilities $\hat{p}_i := c_i/N$. While w.l.o.g. the true probabilities follow $p_1 \geq \ldots \geq p_{|\mathbf{D}|}$, we use the notation $p_{(i)}$ to order by empirical probabilities, so that $\hat{p}_{(1)} \geq \ldots \geq \hat{p}_{(|\mathbf{D}|)}$. Although easy concentration bounds exist when the top-k bins are fixed, in our case the empirical top-k bins are random. Even so we may establish bounds.

Lemma 2. *Choose any $\delta > 0$; with probability at least $1-\delta$,*

$$\Big| \sum_{i=1}^{k} \hat{p}_{(i)} - \sum_{i=1}^{k} p_i \Big| \leq \Delta(\delta) := 4\sqrt{\frac{k^2 \ln(2|\mathbf{D_n}|/\delta)}{2N}}. \quad (5)$$

Given that the top-zk n-grams are selected empirically over $\mathbf{S}$, it follows from Lemma 1 and Lemma 2 that with probability $1-\delta$ these account for $\beta'' := \beta' - \Delta(\delta)$ of the underlying probability measure. Thus, Theorem 1 can be easily adapted for the noise case, as follows.

Theorem 2. *Suppose the observed n-grams are ranked by empirical count and the top k' selected. Let $\beta'' = \beta - \frac{m}{N-m} - \Delta(\delta)$. Then with probability $1-\delta$, equations (3) and (4) hold with β' replaced with β''.*

By setting $k' = zk$ in each result, we can handle the oversampling value z. While for byte sequences the maximum value for $|\mathbf{D}_n|$ is 256^n, the bound in Lemma 2 can be strengthened by filtering out prefixes which are not observed in each prior step.

Experiments

We now demonstrate the superiority of the Intergrams algorithm as compared to other approaches, validating each of our hardware-based observations along the way. As hash-gramming is orders of magnitude faster than any other algorithms (Raff and Nicholas 2018), we compare only against hash-gramming with the 'small tricks' discussed earlier.

Our experimental system is a very powerful single system equipped with 4 32-core Xeon E7-8867 v3 processors, for a total of 128 cores. Each core has 32 KB L1 cache and 512 KB L2 cache. Each processor shares its 45 MB L3 cache across cores, giving a total of 180 MB of L3 cache. The system is equipped with 2 TB of DDR3-1333 RAM, with a disk controller capable of serving data from a RAID at $\sim$5 GB/s.

We implemented the Intergrams algorithm carefully in C++, using the Armadillo library for timing (Sanderson and Curtin 2016). We process sequences in parallel using thread pairs: one thread reads the sequence from disk (this thread is mostly idle, waiting on I/O) while another thread processes

Dataset	Type	Size	m
EMBER[3] (Anderson and Roth 2018)	bytes	1009 GB	800k
C4 (Raffel et al. 2020)	text	751 GB	6.22M
1000gp (Clarke et al. 2012)	genomics	1.4 TB	1.58M

Table 2: Datasets used in our experiments. The *C4* and *1000gp* datasets were chunked such that each sequence contained a few hundred to a few thousand lines of source data.

Algorithm	EMBER ($k = 100$k)			c4 ($k = 10$k)			1000gp ($k = 10$k)		
	Runtime	Speedup	Jaccard	Runtime	Speedup	Jaccard	Runtime	Speedup	Jaccard
`hg-vanilla`	9078.5s	–	–	39787.1s	–	–	10042.0s	–	–
`hg-cuckoo`	8987.9s	1.01x	–	37506.8s	1.06x	–	9682.3s	1.04x	–
`hg-largepage`	8394.7s	1.08x	–	37214.8s	1.06x	–	8640.6s	1.16x	–
`hg-trie`	8286.5s	1.10x	–	33026.1s	1.20x	–	8130.2s	1.24x	–
`hg-fast`	7162.6s	1.27x	–	30811.2s	1.29x	–	7061.5s	1.42x	–
`intergrams`, $z = 1$	1413.7s	**6.42x**	0.71	1183.9s	**33.6x**	0.91	1215.7s	**8.26x**	**1.0**
`intergrams`, $z = 1.5$	1458.8s	**6.22x**	0.91	1373.7s	**29.0x**	**1.0**	1152.9s	**8.71x**	**1.0**
`intergrams`, $z = 2$	1771.4s	**5.13x**	0.97	1501.9s	**26.5x**	**1.0**	1144.6s	**8.77x**	**1.0**

Table 3: Runtime results for Intergrams and variants of the hash-gramming approach. In every case the Intergrams algorithm is able to provide significant speedup (up to 30x!), while still recovering nearly every true top-k n-gram (sometimes exactly!).

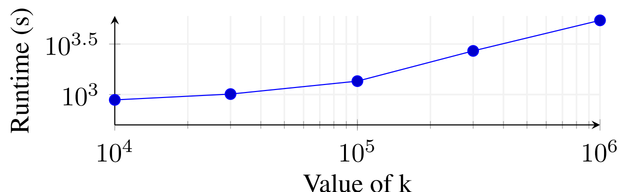

Figure 5: Runtime results for a sweep of k on the EMBER dataset. Y-axis is logarithmic.

data that the other has buffered. This thread pair strategy is necessary because Intergrams is bottlenecked by disk; thus, a thread entirely for I/O ensures maximum disk throughput.

We also implemented hash-gramming carefully in C++; however, due to its poor memory access patterns, it cannot saturate disk bandwidth and thus each sequence is assigned to its own thread, which performs both I/O and the hash-gramming computations. We applied the performance tricks detailed earlier to hash-gramming, producing these variants:

- `hg-vanilla` is hash-gramming implemented as written.
- `hg-largepage` uses 2 MB page sizes to reduce the TLB misses to the global counts array.
- `hg-prefetch` uses software prefetching to mask the latency of fetching the global counts array from RAM.
- `hg-cuckoo` uses cuckoo hashing (Pagh and Rodler 2004) to minimize the updates to the global counts array.
- `hg-fast` uses our tuned trie structure for the second pass, as opposed to the standard C++ `unordered_map`.

The optimizations in each variation are cumulative: that is, `hg-cuckoo` also contains the optimizations from `hg-prefetch` and `hg-largepage`, and so forth. As we designed Intergrams with all these optimizations in mind, ablating them individually does not make sense. Instead, to control the accuracy of Intergrams, we take $z \in \{1, 1.5, 2\}$.

For datasets, we used three large datasets from different application areas, detailed in Table 2. Our implementations were tuned to byte data, and so we processed the text and genomic data at the character (byte) level. Note that for both plaintext (ASCII) and genomics data, the size of $|D|$ is smaller than 256^n, meaning that additional optimization implementations could be made (see the trie discussion).

Step	Runtime	Throughput
3-gram pass	228.2s	4.42 GB/s
top-zk 3-grams/trie building	3.25s	–
4-gram pass	205.95s	4.90 GB/s
top-zk 4-grams/trie building	0.97s	–
5-gram pass	189.37s	5.33 GB/s
top-zk 5-grams/trie building	1.16s	–
6-gram pass	190.57s	5.29 GB/s
top-k 6-grams	0.29s	–

Table 4: Runtime breakdown for each step of the Intergrams algorithm on EMBER with k set to 10k and $z = 1.5$.

Table 3 contains runtime results for each of the datasets. Intergrams *significantly* outperforms all variants of hash-gramming, even the most optimized. In the best cases, Intergrams provides 20–30x speedup due to its hardware-informed design. In addition, even with $z = 1$, it returns n-grams that match those returned by hash-gramming.

Next, we consider the effect of increasing k on the runtime. Figure 5 shows each algorithm's runtime on the EMBER dataset when k is swept from 1k to 1M. Jaccard similarities were comparable to what is seen in Table 3.

Lastly, to show the relative cost of each step of the Intergrams algorithm, Table 4 shows the breakdown of each step of the Intergrams algorithm on each of the three datasets. Since no trie lookup is necessary for the 3-gram pass, that step is the most efficient and can maximize disk throughput. The top-zk steps and trie building steps are negligible compared to the time it takes to pass over the data. In later passes of the algorithm, the trie gets more selective (as compared to $|D|$) and this phenomenon accounts for the 5-gram and 6-gram passes being faster than the 4-gram pass.

Conclusion

We introduced the Intergrams algorithm, a fast algorithm for computing the top-k n-grams on very large datasets. Its empirical performance beats all known other algorithms, and it has favorable theoretical guarantees. Code can be found at `github.com/rcurtin/Intergrams`. We are exploring further improvement by exploiting the Zipf distribution to skip data for a slight loss in accuracy, as in Raff et al. (2025).

References

Anderson, H. S.; and Roth, P. 2018. Ember: an open dataset for training static pe malware machine learning models. *arXiv preprint arXiv:1804.04637*.

Brett, B. 2016. Memory Performance in a Nutshell. Technical Report 672679, Intel Corporation.

Charikar, M.; Chen, K.; and Farach-Colton, M. 2002. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, 693–703. Springer.

Chor, B.; Horn, D.; Goldman, N.; Levy, Y.; and Massingham, T. 2009. Genomic DNA k-mer spectra: models and modalities. *Genome biology*, 10(10): R108.

Clarke, L.; Zheng-Bradley, X.; Smith, R.; Kulesha, E.; Xiao, C.; Toneva, I.; Vaughan, B.; Preuss, D.; Leinonen, R.; Shumway, M.; et al. 2012. The 1000 Genomes Project: data management and community access. *Nature Methods*, 9(5): 459–462.

Cormode, G.; and Muthukrishnan, S. 2005. Summarizing and mining skewed data streams. In *Proceedings of the 2005 SIAM International Conference on Data Mining*, 44–55. SIAM.

Demaine, E. D. 2002. Cache-oblivious algorithms and data structures. *Lecture Notes from the EEF Summer School on Massive Data Sets*, 8(4): 1–249.

Drepper, U. 2007. What every programmer should know about memory. Technical report, Red Hat, Inc.

Fredkin, E. 1960. Trie memory. *Communications of the ACM*, 3(9): 490–499.

Frigo, M.; Leiserson, C. E.; Prokop, H.; and Ramachandran, S. 1999. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*, 285–297. IEEE.

Fuyong, Z.; and Tiezhu, Z. 2017. Malware detection and classification based on n-grams attribute similarity. In *2017 IEEE international conference on computational science and engineering (CSE) and IEEE international conference on embedded and ubiquitous computing (EUC)*, volume 1, 793–796. IEEE.

Intel Corporation. 2023. Intel® 64 and IA-32 Architectures Optimization Reference Manual Volume 1. Technical Report 671488, Intel Corporation.

Jin, C.; Qian, W.; Sha, C.; Yu, J. X.; and Zhou, A. 2003. Dynamically maintaining frequent items over a data stream. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, 287–294.

Joyce, R. J.; Miller, G.; Roth, P.; Zak, R.; Zaresky-Williams, E.; Anderson, H.; Raff, E.; and Holt, J. 2025. EMBER2024 - A Benchmark Dataset for Holistic Evaluation of Malware Classifiers. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, KDD '25, 5516–5526. New York, NY, USA: Association for Computing Machinery. ISBN 979-8-4007-1454-2.

Liu, J.; Min, S.; Zettlemoyer, L.; Choi, Y.; and Hajishirzi, H. 2024. Infini-gram: Scaling Unbounded n-gram Language Models to a Trillion Tokens. In *Proceedings of the First Conference on Language Modeling (COLM 2024)*.

Lu, F.; Curtin, R. R.; Raff, E.; Ferraro, F.; and Holt, J. 2024. High-Dimensional Distributed Sparse Classification with Scalable Communication-Efficient Global Updates. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2037–2047.

Lu, F.; Curtin, R. R.; Raff, E.; Ferraro, F.; and Holt, J. 2025. Optimizing the Optimal Weighted Average: Efficient Distributed Sparse Classification. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 147–163. Springer.

Masud, M. M.; Khan, L.; and Thuraisingham, B. 2008. A scalable multi-level feature extraction technique to detect malicious executables. *Information Systems Frontiers*, 10(1): 33–45.

Metwally, A.; Agrawal, D.; and El Abbadi, A. 2005. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory*, 398–412. Springer.

Pagh, R.; and Rodler, F. F. 2004. Cuckoo hashing. *Journal of Algorithms*, 51(2): 122–144.

Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; et al. 2011. Scikit-learn: Machine learning in Python. *The Journal of Machine Learning Research*, 12: 2825–2830.

Piskorski, J.; and Jacquet, G. 2020. TF-IDF character N-grams versus word embedding-based models for fine-grained event classification: A preliminary study. In *Proceedings of the Workshop on Automated Extraction of Socio-political Events from News 2020*, 26–34.

Raff, E.; Curtin, R. R.; Everett, D.; Joyce, R. J.; and Holt, J. 2025. Zipf-Gramming: Scaling Byte N-Grams Up to Production Sized Malware Corpora. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management*, 5988–5996.

Raff, E.; Fleming, W.; Zak, R.; Anderson, H.; Finlayson, B.; Nicholas, C. K.; Mclean, M.; Fleming, W.; Nicholas, C. K.; Zak, R.; and Mclean, M. 2019. KiloGrams: Very Large N-Grams for Malware Classification. In *Proceedings of KDD 2019 Workshop on Learning and Mining for Cybersecurity (LEMINCS'19)*.

Raff, E.; and Nicholas, C. 2018. Hash-grams: Faster n-gram features for classification and malware detection. In *Proceedings of the ACM Symposium on Document Engineering 2018*, 1–4.

Raff, E.; Zak, R.; Cox, R.; Sylvester, J.; Yacci, P.; Ward, R.; Tracy, A.; McLean, M.; and Nicholas, C. 2018. An investigation of byte n-gram features for malware classification. *Journal of Computer Virology and Hacking Techniques*, 14: 1–20.

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

Sanderson, C.; and Curtin, R. 2016. Armadillo: a template-based C++ library for linear algebra. *The Journal of Open Source Software*, 1(2): 26–26.

Woods, D. R.; and Lyon, J. M. 1973. The INTERCAL Programming Language Reference Manual.

Yang, Z.; Li, H.; Jia, Y.; Zheng, Y.; Meng, H.; Bao, T.; Li, X.; and Luo, L. 2020. Intrinsic laws of k-mer spectra of genome sequences and evolution mechanism of genomes. *BMC Evolutionary Biology*, 20(1): 157.