Hash-Grams: Faster N-Gram Features for Classification and Malware Detection

Edward Raff
Laboratory for Physical Sciences
edraff@lps.umd.edu
Booz Allen Hamilton
raff_edward@bah.com

Charles Nicholas
Univ. of Maryland, Baltimore County
nicholas@umbc.edu

ABSTRACT

N-grams have long been used as features for classification problems, and their distribution often allows selection of the top-k occurring n-grams as a reliable first-pass to feature selection. However, this top-k selection can be a performance bottleneck, especially when dealing with massive item sets and corpora. In this work we introduce Hash-Grams, an approach to perform top-k feature mining for classification problems. We show that the Hash-Gram approach can be up to three orders of magnitude faster than exact top-k selection algorithms. Using a malware corpus of over 2 TB in size, we show how Hash-Grams retain comparable classification accuracy, while dramatically reducing computational requirements.

ACM Reference Format:
Edward Raff and Charles Nicholas. 2018. Hash-Grams: Faster N-Gram Features for Classification and Malware Detection. In *DocEng '18: ACM Symposium on Document Engineering 2018, August 28–31, 2018, Halifax, NS, Canada.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3209280.3229085

1 INTRODUCTION

We define a character n-gram as a set of n consecutive characters that appear in a document. (Word n-grams, defined analogously, have also been studied, but are not the focus of this paper.) It turns out that character n-grams have been used in document analysis for many years [4]. In this paper, we present a faster form of n-gram analysis, with the goal of improving the speed and accuracy of malware classification. Byte n-grams have been used in the study of malicious binaries [8, 14, 17], but we (and others) have noted the difficulty involved in scaling to larger corpora [12, 13].

There are a number of factors that make n-grams difficult to scale in the malware classification use-case. We are concerned with a classification problem where N is exceedingly large, and the documents in question are represented with binary feature-vectors $x \in \{0,1\}^D$ indicating the presence of absence of an n-gram. While one could use other weighting schemes, such as a frequency distribution of some subset of n-grams, it is often the case that the pattern of occurring features is more important than feature magnitude in sparse and high-dimensional spaces [15].

As an example of the scaling problem, let's assume modest value of n=4. With byte values in the range 0-255, the number of possible n-grams is 256^4 or about four billion. To build the binary feature vector mentioned above for an input file of length D, n-grams would need to be inspected, and a certain bit in the feature vector set accordingly. Malware specimen n-grams do tend to follow a Zipfian distribution [13, 18]. Even so, we still have to deal with a massive "vocabulary" of n-grams that is too large to keep in memory. It has been found that simply selecting the top-k most frequent n-grams, which will fit in memory, results in better performance than a number of other feature selection approaches [13]. For this reason we seek to solve the top-k selection faster with fixed memory so that n-gram based analysis of malware can be scaled to larger corpora.

To tackle this problem we keep track of the top-k *hashed* n-grams, and simply ignore collisions. By using a large amount of RAM, we can obtain significantly better runtime performance compared to exact counting of n-grams (which requires out-of-core processing) or other top-k item mining approaches while also proving that we will select the top-k hashed n-grams with high probability. We show empirically that this works well on a large corpus of 2 TB of compressed data, is orders of magnitude faster than the naive approach, and show empirically that our approach has 100% recall of top-k hashes in practice.

1.1 Related Work

There exists two groups of work related to our own based on hashing and frequent item-set mining. In the hashing case, a method known as the hashing-trick [16] is closely related. The hashing-trick maps features to indices in a sparse feature vector. This can be exploited for working with n-grams to avoid having to count all occurring n-grams and find the most likely ones. It works by mapping D values for a datum x into a new m dimensional space, where m is a hyper parameter. Any feature processing / selection is normally done before hashing.

While useful, the variance of the hashing trick in our case of $x \in \{0,1\}^D$ would be $\sigma^2 = O(m^{-1}D^2)$ [16], and thus wouldn't work well without making $m >> D$. Beyond a computational requirement in RAM, this makes the learning problem harder due to the curse of dimensionality [6]. In our approach we will use a large amount of RAM to select the top-k items, resulting in a more reasonable dimension size since $k << D$.

Since not all n-grams can be kept in memory, unless n is quite small, another approach is to tackle the problem by selecting the top-k most frequent items from a stream [2, 3, 7, 9]. Such approaches develop data structures that attempt to adaptively drop infrequent items as memory becomes constrained and work well under Zipfian distributed data. While effective, the added overhead of these structures makes them slower and difficult to parallelize.

DocEng '18, August 28–31, 2018, Halifax, NS, Canada

ACM ISBN 978-1-4503-5769-2/18/08…$15.00
https://doi.org/10.1145/3209280.3229085

2 HASH-GRAMMING

Our hash-gramming approach begins with the creation of a large array of size B to count the number of times each n-gram occurred, and use a hash function $h(\cdot)$ to map each n-gram to a location in the array. We then iterate through all documents and increment counters, using atomic updates to handle parallelism. The entire procedure is given in Algorithm 1. Once done, we use Quick-select algorithm to find the top k counts in $O(B)$ time [5]. If relative ordering is desired, the k selected n-grams can then be sorted.

Algorithm 1 Hash-Graming

Require: Bucket size B, rolling hash function $h(\cdot)$, corpus of S documents, and desired number of frequent hash-grams k.

1: T ← new integer array of size B
2: **for** all documents $x \in S$ **do** ▷ $O(L)$ for L total n-grams
3: **for** n-gram $g \in x$ **do**
4: $q' \leftarrow h(q) \mod B$
5: $T[q'] \leftarrow T[q'] + 1$ ▷ Update atomically if using multiple threads
6: **end for**
7: **end for**
8: $T_k \leftarrow$ QuickSelet(T, k) ▷ $O(B)$
9: Sort T_k ▷ Option if we desired ranking, $O(k \log k)$
10: **return** T_k in order

Given a rolling hash-function (one which returns the hash of a moving window) and L total observed n-grams to evaluate, the entire procedure takes $O(L+B)$ time while being simple to implement. This is asymptotically equivalent to frequent item-set approaches like the Space Saving algorithm [9], but has significantly less overhead. To extract features from new data items at test time, one simply hashes the n-grams and checks if they are in the k kept hashes.

Our implementation is in Java, and so (unless otherwise specified) we use a value of $B = 2^{31} - 18$, the largest prime smaller than Java's maximum integer value of $2^{31} - 1$. Choosing a prime value ensures a uniform distribution of hash values on line 4. Assuming the data follows a power-law distribution, as n-grams in executable binaries usually do, we expect Algorithm 1 to retain all of the hashes (or "hash-grams") that belong to the top-k most frequent n-grams — something we will show in subsection 2.1.

Because there are often more total n-grams than bucket locations B, the Pigeonhole Principle implies that a number of infrequent n-grams will collide with the top-k hashes and each other. However the intrinsic nature of the infrequent n-grams colliding with the top-k n-grams means that the infrequent ones will have little if any impact on the accuracy of out results.

2.1 Hash-Gramming under the Zipfian Distribution

As noted above, it is often the case that the frequency of n-grams in binaries follows a Zipfian power-law distribution. Under this assumption, we can show that the Hash-Gram approach is unlikely to ever miss a top-k n-gram. First, recall that the Zipfian distribution for an alphabet of N possible items has the Probability Mass Function (PMF) (1), where $x \in [1, N]$ is the rank of each feature.

$$f(x; p, N) = \begin{cases} \frac{x^{-p-1}}{H_N^{(p+1)}} & 1 \le x \le n \\ 0 & \text{else} \end{cases} \tag{1}$$

$H_N^{(p)} = \sum_{i=1}^{N} 1/i^p$ indicates the N'th harmonic number of the p'th order. The cumulative distribution function (CDF) F is given by

$$F(x; p, N) = \begin{cases} \frac{H_x^{(p+1)}}{H_N^{(p+1)}} & 1 \le x \le n \\ 1 & x > n \end{cases} \tag{2}$$

The set of N possible items corresponds to our n-grams over some alphabet. Determining if an infrequent n-gram can incorrectly make it into the top-k hash-grams means that enough n-grams need to collide into a single bucket B_i (based on the hash function $h(\cdot)$) such that their count will be greater than one of the true top-k items. Further, the infrequent n-grams that collide into the same bucket as a frequent n-gram reinforce the frequent n-gram's selection.

Let b_t indicate the bucket to which a top-k n-gram was hashed, and let b_b be a bucket that has none of the top-k n-grams. To assess the likelihood of a top-k n-gram's selection, we need to answer the question: what is the probability of the difference between the non-top-k n-grams in buckets b_t and b_b is larger than the frequency of the k'th top n-gram? Because we want to compare the frequency counts, and $\text{Freq}(x) \propto f(x; p, N)$, we will use PMF itself as the relative frequency.

For simplicity we will assume that the infrequent n-grams are uniformly distributed between all buckets. Given a total of B buckets, each bucket will have L/B infrequent n-grams colliding. We can also describe the difference between two buckets as a random variable $Z = a - b$, where $a, b \sim \text{Freq}(\text{Zipf}^k(x; p, N))$ and $a \perp\!\!\!\perp b$.

We use $\text{Zipf}^k(x; p, N) = f(x; p, N)/(1 - F(k; p, N))$ to denote the truncated distribution that removes the top-k most frequent items from consideration, as we are assuming that the top-k items have already been placed in buckets.

$$\mathbb{P}(Z \ge \mathbb{E}[Z] + t) \le \frac{\text{Var}(Z)}{t^2} \tag{3}$$

What is the probability that the sum of L/B samples of Z will be greater than than the k'th most frequent n-gram? To answer this question, we can use Chebyshev's inequality (3)

$$\mathbb{P}\left(\sum_{i=1}^{L/B} Z_i \ge t\right) \le \frac{2\text{Var}(\text{Freq}(\text{Zipf}^k(p, N)))}{(L/B)(tB/L)^2}$$

Since we use the PMF's value as the proportional frequency under the power-law assumption, we want $t = f(k; p, N)$, and the proportional frequency sum $\sum_i Z_i$ to be a small value. With some calculation we find that the mean truncated frequency is (4)

$$\mathbb{E}[\text{Freq}(\text{Zipf}^k(x; p, N))] = \frac{H_N^{(p+1)} H_N^{(3p+3)} - \left(H_N^{(2p+2)}\right)^2}{\left(H_N^{(p+1)}\right)^4} \tag{4}$$

Which we use to get the variance (5), where $\zeta(s, a) = \sum_{i=0}^{\infty} 1/(s+i)^a$ is the Hurwitz Zeta function. Using this we can then derive $\text{Var}(\text{Freq}(\text{Zipf}^k(p, N)))$, as given in (5).

$$- \frac{\left(H_k^{(p+1)} - H_N^{(p+1)}\right) \cdot \left(\zeta(3p+3,k) - \zeta(3p+3,N+1)\right) + \left(\zeta(2p+2,k) - \zeta(2p+2,N+1)\right)^2}{\left(H_N^{(p+1)}\right)^2 \left(H_k^{(p+1)} - H_N^{(p+1)}\right)^2} \quad (5)$$

Which allows us to arrive at the final bound (6)

$$\mathbb{P}\left(\sum_{i=1}^{L/B} Z_i \geq f(k;p,N)\right) \leq \frac{4k^{2p+2}\left(\left(H_k^{(p+1)} - H_N^{(p+1)}\right)\left(\zeta(2p+2,k) - \zeta(2p+2,N+1)\right)^2 + \left(H_k^{(p+1)} - H_N^{(p+1)}\right)\left(\zeta(3p+3,k) - \zeta(3p+3,N+1)\right)\right)^2}{B \cdot L \left(H_N^{(p+1)}\right)^2 \left(H_k^{(p+1)} - H_N^{(p+1)}\right)^4} \quad (6)$$

Despite the intimidating look of (6), it tells us that it is *quite* unlikely for a top-k hash-gram to not be selected. As an example, we will consider the case of the n-gram model from [12], which used byte 6-grams on over 2 TB of data. Here $N = 256^6$, $L \approx 5 \cdot 10^5$ and $k = 100,000$. The worse case scenario of $p = 1$ gives us a probability $\leq 5.4 \cdot 10^{-44}$. Thus we can expect to obtain the correct top-k hashes with high probability, even under pessimistic assumptions of the power-law coefficient p.

3 EXPERIMENTAL RESULTS

Having described our Hash-Gram algorithm, and provided theoretical results backing its use, we move on to demonstrate its utility in experimental evaluation. First we will show the Hash-Gram's speed advantage over pre-existing techniques like the Space-Saving algorithm on synthetic Zipfian data. Then we will show it obtains comparable results at classification compared to exact n-gramming of a corpus for malware detection. The Space-Saving algorithm is designed for selecting from streams, and guarantees selecting the true top-k if $k << B$ [1].

3.1 Synthetic Results

For our first test we generate data from the Zipfian distribution with $N = 2^{31} - 1$ and $p = 1$. We then compare how many items-per-second can be added to the Space-Saving algorithm[1] and to our new Hash-Gram structure as we vary the number of buckets B, and set the desired number of n-grams $k = B/100$.

Because the true rank ordering of the Zipfian data is known from its definition in Equation 1, we are able to easily check that both approaches have 100% recall for every test case. This empirically confirms our theoretical results in subsection 2.1.

The timing results can be found in Figure 1, and show the average number of items-per-second over 10 different runs. When the bucket size is small and presents little overhead, both approaches show near-constant runtime for $B \in [10^3, 10^6]$. In this range Hash-Grams are consistently 8.4 times faster than the Space-Saving approach. However, after this point the Space-Saving algorithm begins to degrade. By $B = 262,144,000$, the Hash-Gram approach has become 788 times faster compared to the Space-Saving algorithm.

[1]Code obtained from https://github.com/fzakaria/space-saving

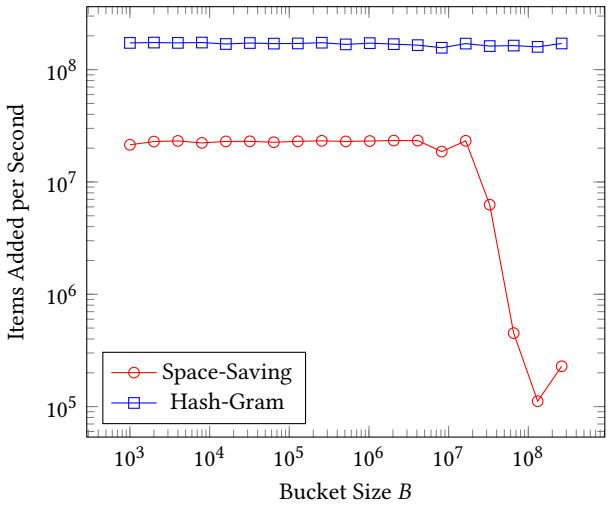

Figure 1: Plot shows how many items-per-second (y-axis, higher is better) can be added to the Hash-Gram and Space-Saving structures when using a single thread. Bucket count is on the x-axis. Note the log-log scale.

We believe these runtime differences emerge as an artifact of inconsistent memory access patterns over time for the Space-Saving approach, compounded by the inconsistent amount of work that must be done. The Hash-Gram will also have inconsistent memory accesses, but every access will involve the same number of operations. That Hash-Grams require only a single object, rather than multiple indirections, also works to its advantage.

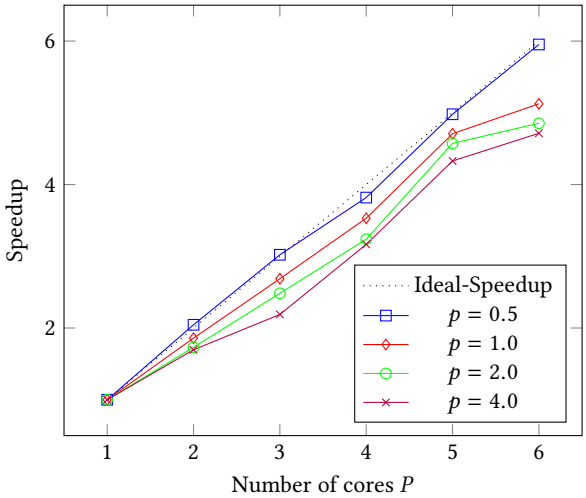

Figure 2: Speedup of the Hash-Gram approach as more cores P are used. Tested with synthetic Zipf data using $N = 2^{31} - 1$, and varying skweness p.

As mentioned, the Hash-Gram approach is also easy to parallelize — one simply makes the increments to counters using atomic operations. This allows one to obtain near-linear speedups as more

cores are used, as shown in Figure 2 on up to $P = 6$ CPU cores. Here the Hash-Gram was run for multiple Zipfian distributions with different skewness p. Speedup is impacted negatively as the skew increases, as it means the most frequent items will increase their dominance, and thus increase the contention for atomic updates.

While the Space-Saving algorithm can be parallelized, it is more involved to do so [1]. In contrast to our Hash-Graming, the Space-Saving approach has better scalability with P as the skewness increases, as it means less updates to the Space-Saving data structure.

3.2 Malware Detection Results

We've now shown on synthetic data that the Hash-Gram approach has the empirically perfect recall of all top-k hashed n-grams, parallelizes efficiently, and scales better than the Space-Saving approach. The only remaining question is how accurate are models built from hash-grams, as opposed to exact top-k n-gram selection? We test this with a malware detection task, involving the 2 million binaries used for training a malware detection model from [11, 12] and trained in the same style. We compare to the same highly optimized n-gram counting code that ran on a cluster of 12 machines for two weeks. For the Hash-Gram model, we use one machine from the same cluster to perform n-gram extraction. Then we trained an Elastic-Net regularized [19] Logistic Regression mode using JSAT [10].

Table 1: Accuracy, AUC, and compute time for N-Gram and Hash-Gram features on 2 million Windows executables.

	Industry		Public		
	Acc	AUC	Acc	AUC	CPU Hours
N-Grams	91.6	97.0	82.6	93.4	32,256
Hash-Grams	91.2	97.1	83.2	92.9	469

The results are presented in Table 1, where we see that the exact n-gram and hash-gram models have indistinguishable accuracy on both of the test sets. All scores are close, and fluctuated between slightly better and worse on individual numbers. The slight changes in results are not unexpected due to the hash collisions, but is clearly of equivalent predictive quality to the exact n-gram model.

The Hash-Gram approach was 68.8 times faster in extracting the top-k features compared to the n-gram approach, allowing us to reduce a two week job on a cluster down to under three days on a single node. We note that the code used for the exact n-gram is highly optimized Java code that has gone through three years of performance tuning and improvements to scale up the n-gram processing.

The primary weakness of the Hash-Gram approach is the ability to inspect the selected n-grams themselves, which have been lost by the hashing process. This prevents any analysis that requires inspection of the original n-grams, as was done in [13].

4 CONCLUSION

We have developed Hash-Graming, a new and simple method to mining frequent n-grams for classification tasks. It is easy to implement, parallelize, orders of magnitude faster than previous tools, and comes with provable bounds on obtaining the top-k hashed n-grams. We've shown on a challenging malware detection problem that models trained on hash-grams retain the predictive performance of exact n-grams.

REFERENCES

[1] Massimo Cafaro, Marco Pulimeno, Italo Epicoco, and Giovanni Aloisio. 2018. Parallel space saving on multi- and many-core processors. *Concurrency and Computation: Practice and Experience* 30, 7 (4 2018). https://doi.org/10.1002/cpe.4160

[2] Graham Cormode and Marios Hadjieleftheriou. 2008. Finding Frequent Items in Data Streams. *Proc. VLDB Endow.* 1, 2 (8 2008), 1530–1541. https://doi.org/10.14778/1454159.1454225

[3] Graham Cormode and S. Muthukrishnan. 2005. Summarizing and mining skewed data streams. In *Proc. of the 2005 SIAM International Conference on Data Mining*. 44–55. https://doi.org/10.1137/1.9781611972757.5

[4] Marc Damashek. 1995. Gauging Similarity with N-Grams. *Science* 267, 5199 (1995), 843–848.

[5] Dorit Dor and Uri Zwick. 2001. Median Selection Requires $(2+\epsilon)$N Comparisons. *SIAM J. Discret. Math.* 14, 3 (3 2001), 312–325. https://doi.org/10.1137/S0895480199353895

[6] Piotr Indyk and Rajeev Motwani. 1999. Approximate Nearest Neighbors : Towards Removing the Curse of Dimensionality 1 Introduction. *Young* (1999).

[7] Cheqing Jin, Weining Qian, Chaofeng Sha, Jeffrey X Yu, and Aoying Zhou. 2003. Dynamically Maintaining Frequent Items over a Data Stream. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management (CIKM '03)*. ACM, New York, NY, USA, 287–294. https://doi.org/10.1145/956863.956918

[8] J Zico Kolter and Marcus A Maloof. 2006. Learning to Detect and Classify Malicious Executables in the Wild. *Journal of Machine Learning Research* 7 (12 2006), 2721–2744. http://dl.acm.org/citation.cfm?id=1248547.1248646

[9] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *Proceedings of the 10th International Conference on Database Theory (ICDT'05)*. Springer-Verlag, Berlin, Heidelberg, 398–412. https://doi.org/10.1007/978-3-540-30570-5{_}27

[10] Edward Raff. 2017. JSAT: Java Statistical Analysis Tool, a Library for Machine Learning. *Journal of Machine Learning Research* 18, 23 (2017), 1–5. http://jmlr.org/papers/v18/16-131.html

[11] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. 2017. Malware Detection by Eating a Whole EXE. *arXiv preprint arXiv:1710.09435* (10 2017). http://arxiv.org/abs/1710.09435

[12] Edward Raff and Charles Nicholas. 2017. Malware Classification and Class Imbalance via Stochastic Hashed LZJD. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security (AISec '17)*. ACM, New York, NY, USA, 111–120. https://doi.org/10.1145/3128572.3140446

[13] Edward Raff, Richard Zak, Russell Cox, Jared Sylvester, Paul Yacci, Rebecca Ward, Anna Tracy, Mark McLean, and Charles Nicholas. 2016. An investigation of byte n-gram features for malware classification. *Journal of Computer Virology and Hacking Techniques* (9 2016). https://doi.org/10.1007/s11416-016-0283-1

[14] M.G. Schultz, E. Eskin, F. Zadok, and S.J. Stolfo. 2001. Data Mining Methods for Detection of New Malicious Executables. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. IEEE Comput. Soc, 38–49. https://doi.org/10.1109/SECPRI.2001.924286

[15] Anshumali Shrivastava and Ping Li. 2014. In Defense of Minhash over Simhash. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics, AISTATS ([JMLR} Workshop and Conference Proceedings)*, Vol. 33. JMLR.org, Reykjavik, Iceland, 886–894. http://jmlr.org/proceedings/papers/v33/shrivastava14.html

[16] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. 2009. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09*. ACM Press, New York, New York, USA, 1113–1120. https://doi.org/10.1145/1553374.1553516

[17] Richard Zak, Edward Raff, and Charles K. Nicholas. 2017. What Can N-Grams Learn for Malware Detection?. In *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 109–118.

[18] George Kingsley Zipf. 1949. *Human behavior and the principle of least effort*. Addison-Wesley Press, Oxford, England. xi, 573–xi, 573 pages.

[19] Hui Zou and Trevor Hastie. 2005. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society, Series B* 67, 2 (4 2005), 301–320. https://doi.org/10.1111/j.1467-9868.2005.00503.x