Fast Algorithms for Learning with Long N-grams via Suffix Tree Based Matrix Multiplication

Hristo S. Paskov
Computer Science Dept.
Stanford University
hpaskov@stanford.edu

John C. Mitchell
Computer Science Dept.
Stanford University
john.mitchell@stanford.edu

Trevor J. Hastie
Statistics Dept.
Stanford University
hastie@stanford.edu

Abstract

This paper addresses the computational issues of learning with long, and possibly all, N-grams in a document corpus. Our main result uses suffix trees to show that N-gram matrices require memory and time that is at worst *linear* (in the length of the underlying corpus) to store and to multiply a vector. Our algorithm can speed up any N-gram based machine learning algorithm which uses gradient descent or an optimization procedure that makes progress through multiplication. We also provide a linear running time and memory framework that screens N-gram features according to a multitude of statistical criteria and produces the data structure necessary for fast multiplication. Experiments on natural language and DNA sequence datasets demonstrate the computational savings of our framework; our multiplication algorithm is four orders of magnitude more efficient than naïve multiplication on the DNA data. We also show that prediction accuracy on large-scale sentiment analysis problems benefits from long N-grams.

1 Introduction

N-gram models are indespensible in natural language processing (NLP) , information retrieval, and, increasingly, computational biology. They have been applied succussfully in sentiment analysis (Paskov, 2013), text categorization (Cavnar, 1994), author identification (Houvardas, 2006), DNA function prediction (Kähärä, 2013), and numerous other tasks. The allure of N-gram models comes from their simplicity, interpretability, and efficacy: a document corpus is represented by its N-gram matrix, where each row/column corresponds to a distinct document/N-gram respectively, and each entry counts the number of occurrences of that N-gram in the document. The N-gram matrix provides a feature representation for statistical modelling and the coefficients of each N-gram can often be interpreted as a score indicating their relevance to the task.

At the simplest extreme, unigrams provide a summary of the word distribution in each document and serve as an effective baseline representation for a variety of NLP tasks. Higher order N-grams provide more nuance by capturing short-term positional information and can achieve state of the art results on a variety of tasks (Wang, 2012) (Paskov, 2013). A canonical example of the value of longer N-grams is given by the phrase "I don't like horror movies, but this was excellent," which fails to convey its positive sentiment when its words are scrambled. Unfortunately, this additional information comes at a cost: a document of n words may contain up to $\Theta(Kn)$ *distinct* N-grams of length K[1]. This growth makes the memory and computational burden of training N-gram models beyond bigrams impractical for large natural language corpora. Statistically, these larger feature representations suffer from the curse of dimensionality (Hastie, 2001) and may lead the model to overfit, so careful regularization is necessary.

This paper ameliorates the computational burden of learning with long N-grams. We demonstrate how the structure of suffix trees can be used to store and multiply[2] any N-gram matrix in time and space that is at most *linear* in the length of the underlying corpus. As most learning algorithms rely on matrix-vector multiplication to learn and predict, our results equate the computational cost of learning with N-gram matrices to scanning through the original corpus. Our method can speed up *any* learning algorithm that exhibits such structure by simply replacing its multiplication routine with ours. Fast multiplication is possible by means of a specialized data structure that efficiently represents the algebraic structure of the N-gram matrix. In view of the statistical issues associated with long N-grams, we provide a linear running time and memory framework that not only computes this data structure, but also filters N-grams by various criteria and outputs necessary column

[1] We use N-grams of length K to mean N-grams of length *at most* K for brevity.

[2] Multiplication always refers to matrix-vector multiplication.

normalizations. The emphasis of this framework is minimality: by only storing the topological structure of the suffix tree we achieve memory requirements that are comparable to storing the original document corpus. As such, our framework can be used to permanently store the corpus in a format that is optimized for machine learning.

Our paper is organized as follows: section 2 derives the fast multiplication algorithm by showing that after redundant columns in the N-gram matrix are removed, the algebraic structure of the resulting submatrix is encoded by the suffix tree of the underlying corpus. We then investigate how this matrix can be used as a black-box in various common learning scenarios in section 3. Section 4 presents our preprocessing framework. Timing and memory benchmarks that demonstrate the efficacy of the multiplication algorithm are presented in section 5. We also find that high-order N-grams can improve prediction accuracy in large-scale sentiment analysis tasks.

1.1 Related Work

Suffix trees and arrays are used by (Vish., 2004), (Teo, 2006), and (Rieck, 2008) for kernels that efficiently compute pair-wise document similarities based on N-grams. Computing the similarity of all document pairs limits kernels to moderately sized datasets and the lack of explicit features prevents the use of sparsity inducing regularizers such as in the Lasso (Tibs., 1996). Next, (Zhang, 2006) use suffix trees to identify useful N-grams in a text corpus and to show that the all N-gram matrix may be pruned since it contains redundant columns. We show in section 2 that the resulting N-gram matrix may still have too many entries to be practical for large corpora and observe this experimentally. Suffix trees are also used by (Wood, 2011) to efficiently represent and perform inference with a hierarchical process for text. Finally, while (Abou., 2004) and (Kasai, 2001) provide space efficient frameworks for working with suffix arrays, our framework is specialized to statistical processing and achieves greater memory efficiency.

1.2 Multiplication in Machine Learning

We briefly discuss the importance of matrix-vector multiplication for learning. Let $\mathbf{x}_1, \ldots, \mathbf{x}_N \in \mathbb{R}^d$ be N data points with corresponding labels $y_1, \ldots, y_N \in \mathcal{Y}$ and let $X \in \mathbb{R}^{N \times d}$ be the feature matrix that stores $\mathbf{x}_i$ as its i^{th} row. Matrix-vector multiplication operations abound in all phases of supervised and unsupervised learning: basic preprocessing that computes normalizings factors of the form $X^T\mathbf{1}$ (or $X\mathbf{1}$) for every feature (or data point); screening rules that use $|X^T y|$ (when $\mathcal{Y} \subset \mathbb{R}$) to exclude uninformative features (Tibs., 2010); or predictions of the form $f(Xw)$ where w is a learned vector of weights.

Multiplication is also essential for many of the optimization techniques that lie at the core of these learning algorithms. A variety of learning problems can be expressed as optimization problems of the form

$$\underset{w \in \mathbb{R}^d, \beta \in \mathbb{R}^p}{\text{minimize}} \; L_y(Xw, \beta) + \lambda R(w) \tag{1}$$

where w, β are the learning parameters, L_y is a loss function that encodes the y_i labels (if the problem is supervised), and R is a regularization penalty. It is important to remember that this framework captures a number of *unsupervised* learning problems as well, such as Principle Component Analysis, which is useful directly and as a preprocessing step for clustering, deep learning, and other techniques (Hastie, 2001). Any (sub)gradient[3] of (1) with respect to w is given by

$$g_w \in X^T \partial_{Xw} L(Xw, \beta) + \lambda \partial_w R(w). \tag{2}$$

where $\partial_z f(z)$ is the subdifferential of f with respect to z.

Since every (sub)gradient descent method (Parikh, 2013) or accelerated variant critically relies on g_w as a search direction, computing Xw and then $X^T[\partial_{Xw} L(Xw, \beta)]$ is essential and often the most costly part of the optimization. A number of other popular large-scale optimization methods also reduce to multiplying X repeatedly. These include Krylov subspace algorithms such as the conjugate gradient method, and various quasi-Newton methods including BFGS and its limited memory variant (Nocedal, 2006).

1.3 Background and Notation

Let Σ be a finite vocabulary with a strict total ordering $\prec$ over its elements. A *document* $D = x_1 x_2 ... x_n$ of length n is a list of n characters drawn from Σ and an N-gram is any string of characters drawn from Σ. We will refer to each of the n suffixes in D via $D[i] = x_i x_{i+1} ... x_n$. We denote the set of all substrings in D by $D^* = \{\, x_i ... x_{i+k} \mid 1 \le i \le n, 0 \le k \le n - i \,\}$ and the set of all substrings in a document corpus of N documents $\mathcal{C} = \{D_1, \ldots, D_N\}$ as $\mathcal{C}^* = \bigcup_{i=1}^{N} D_i^*$.

Given a subset $\mathcal{S} \subseteq \mathcal{C}^*$ of the set of substrings in (any of) the documents, entry X_{is} of the N-gram matrix $X \in \mathbb{Z}_+^{N \times |\mathcal{S}|}$ counts how many times substring $s \in \mathcal{S}$ appears in document D_i. We use M_i to indicate the i^{th} column of matrix M; when each column pertains to a specific mathematical object, such as an N-gram or tree node, we may use that object as an index (to avoid imposing a particular ordering over the objects). We will *always* take X to be an N-gram matrix for an implicitly given corpus.

A *compact tree* $\mathcal{T} = (V, E)$ is a tree with nodes V and edges E where every internal node is required to have at least 2 children. This ensures that if $\mathcal{T}$ has n leaves, then there are at most $n - 1$ internal nodes. We use $\text{ch}(v) \subset V$ and $\text{p}(v) \in V$ to denote the children and parent of $v \in V$,

[3] To handle non-differentiable objectives, see (Parikh, 2013).

respectively. The root node is given by root($\mathcal{T}$), the depth of any node $v \in V$ is d(v) (with d(root($\mathcal{T}$)) $= 1$), and depth(T) is the maximum depth of any node in V. Finally, a *branch* of $\mathcal{T}$ is a path starting at the root and ending at a leaf; we will use the terminal leaf to identify branches. We will also be concerned with subtrees $\hat{\mathcal{T}} = (\hat{V}, \hat{E})$ of $\mathcal{T}$ which contain a subset $\hat{V} \subset V$ of its nodes. We allow the new edge set $\hat{E}$ to be arbitrary and add a second argument to ch$(v, \hat{E})$ and p$(v, \hat{E})$ to indicate that parent/child relationships are taken with respect to this new edge set.

1.3.1 Suffix Trees

Given a document $D = x_1 x_2 ... x_n$ whose characters belong to an alphabet Σ, the *suffix tree* $\mathcal{T}_D = (V, E)$ for D is a compact tree with n leaves, each of which corresponds to a distinct suffix of D and is numbered according to the starting position of the suffix $1, \ldots, n$. The edges along branch i are labelled with non-empty substrings that partition $D[i]$: suffix $D[i]$ can be recovered by concatenating the edge labels from the root to leaf i. Let l(e) for $e \in E$ be the label of edge e and define the *node character* $c(v)$ of any non-root node $v \in V$ to be the first character of l((p(v), v)). The nodes of $\mathcal{T}_D$ are constrained so that siblings may not have the same node character and are ordered according to the $\prec$ relation on these characters. These constraints ensure that every node has at most $|\Sigma|$ children and they allow for well-defined traversals of $\mathcal{T}_D$. Moreover, every substring $s \in D^*$ is represented by a unique path in $\mathcal{T}_D$ that starts at the root node and terminates in — possibly the middle of — an edge. Similarly to suffixes, s equals the concatenation of all characters encountered along edges from the root to the path's terminus (only a prefix of the final edge will be concatenated if the path ends in the middle of an edge).

Remarkably, $\mathcal{T}_D$ can be constructed in $O(n)$ time (Gusfield, 1997) and has n leaves and at most $n - 1$ internal nodes, yet it represents all $O(n^2)$ distinct substrings of D. This is possible because any substrings whose path representation in $\mathcal{T}$ ends at the same edge belong to the same *equaivalence class*. In particular, for $v \in V \backslash \{\text{root}(\mathcal{T}_D)\}$ suppose that edge (p(v), v) has a label $t = x_i \ldots x_{i+k}$ and let s be the string obtained by concatenating the edge labels on the path from root($\mathcal{T}_D$) down to $p(v)$. Then the strings $S(v) = \{sx_i, sx_{i+1}, \ldots, sx_{i+k}\}$ belong to the same equivalence class because they occur in the same locations, i.e. if sx_i starts at location l in D, then so do all members of $S(v)$. For example, in the string "xaxaba" the substrings "x" and "xa" belong to the same equivalence class.

The *generalized suffix tree* $\mathcal{T}_\mathcal{C}$ for a document corpus $\mathcal{C}$ of n words compactly represents the set of all substrings in $\mathcal{C}^*$ and has n leaves pertaining to every suffix of every document in $\mathcal{C}$. Leaves are also annotated with the document they belong to and $\mathcal{T}_\mathcal{C}$ inherits all of the linear-time storage and computational guarantees of the regular suffix tree (with respect to the corpus length n).

1.3.2 Tree Traversals and Storage

The majority of algorithms in this paper can be expressed as a bottom-up or top-down traversal of a tree $T = (V, E)$ (typically the suffix tree or one of its subtrees) in which information is only exchanged between a parent and its children. Given a fixed ordering of V, the necessary information for a traversal is the topology of T, i.e. its parent-child relationships, as well as any node annotations necessary for the computation. We use two formats which efficiently store this information and make traversals easy: the *breadth-first format* (BFF) and *preorder depth-first format* (DFF). In both cases we distinguish between the internal nodes and leaves of T and divide them into their respective sets $I \cup L = V$. In the BFF we order the nodes of I according to their *breadth-first traversal* whereas in the DFF we order the nodes of I according to their *preorder depth first traversal*; both formats assign indices $[0, \ldots, |I|)$ to the nodes in I. Note that for these traversals to be well defined we assume that the children of each node are ordered in some (arbitrary) but fixed manner. Next, the leaves of T, i.e. L, are assigned indices $[|I|, \ldots, |V|)$ so that if $u, v \in L$ and $p(u)$ comes before $p(v)$ – note that both parents must be in I – then u comes before v. This ordering ensures that leaves are ordered into contiguous blocks with respect to their parent and that the blocks are in the same order as I.

A pair of arrays $(\text{ch}^I, \text{ch}^L)$, each of size $|I|$, capture the topology of T: for all $v \in I$, $\text{ch}^I_v = |\text{ch}(v) \cap I|$ stores the number of internal children of v and $\text{ch}^L_v = |\text{ch}(v) \cap L|$ stores the number of leaf children of v. The number of bits needed to store this topology is

$$|I|(\lceil \log_2 U(I) \rceil + \lceil \log_2 U(L) \rceil) \tag{3}$$

where $U(I), U(L)$ are the largest values in ch^I, ch^L respectively, i.e. the largest number of internal/leaf children for any node. Given node annotations in the same order as the BFF or DFF, top-down/bottom-up traversals are easy to perform by a linear sweep of the annotations and ch^I, ch^L arrays. All memory access is sequential and can be performed efficiently by standard (i.e. desktop) memory and processors.

A speed/memory tradeoff exists for the two formats. The amount of random access memory necessary for a traversal is proportional to the *depth* of T for DFF versus the *width* of T for the BFF. As we discuss in section 4, the former is likely to be smaller than the latter for our purposes. The space savings of the DFF are achieved by maintaining a stack of active nodes pertaining to the current branch being processed. The additional logic required for this bookkeeping makes the DFF slightly slower than the BFF for the traversal. As such, the DFF is useful for more complicated computations in which the amount of information stored per node may be large, whereas the BFF is useful for simple computations that will be performed many times.

2 Fast Multiplication

This section presents our fast multiplication algorithm. Let $\mathcal{T}_\mathcal{C} = (V, E)$ be the suffix tree for a document corpus $\mathcal{C} = \{D_1, \ldots, D_N\}$ and let X be an N-gram matrix containing a column for every $s \in \mathcal{S} \subseteq \mathcal{C}^*$, i.e. the N-grams we are interested in. In order to uncover the necessary algebraic structure for our algorithm we must first remove redundant columns in X. As observed in (Zhang, 2006), redundant columns occur whenever strings in $\mathcal{S}$ belong to the same equivalence class. This implies the following lemma:

Lemma 1. *For any $v \in V$, any $s, s' \in \mathcal{S} \cap S(v)$ have the same distribution among the documents in $\mathcal{C}$ so $X_s = X_{s'}$.*

We remove this redundancy by working with the *node matrix* $\mathcal{X} \in \mathbb{Z}_+^{N \times M}$, a submatrix of X that contains a single column for the M equivalence classes present in $\mathcal{S}$. Formally, node $v \in V$ is present in X if $S(v) \cap \mathcal{S} \neq \emptyset$ and we define $\mathcal{V} \subset V$[4] to be the set of all nodes present in X. Column $\mathcal{X}_v$ for $v \in \mathcal{V}$ is obtained by picking an arbitrary $s \in S(v) \cap \mathcal{S}$ and setting $\mathcal{X}_v = X_s$. We can also reconstruct X from $\mathcal{X}$ by replicating column $\mathcal{X}_v$ $|S(v) \cap \mathcal{S}|$ times; this underscores the inefficiency in the N-gram matrix.

2.1 Linear Dependencies in the Node Matrix

We are now ready to show how the topology of $\mathcal{T}_\mathcal{C}$ determines the linear dependencies among the columns of $\mathcal{X}$. Central to our analysis is the lemma below, which shows that the document frequency of any node is determined entirely by the leaves of its subtree:

Lemma 2. *The number of times node $v \in V \backslash \{root(\mathcal{T}_\mathcal{C})\}$ appears in document $D_i \in \mathcal{C}$ equals the number of leaves that belong to D_i in the subtree rooted at v.*

The simplest case occurs when $\mathcal{V} = V \backslash \{\text{root}(\mathcal{T}_\mathcal{C})\}$, i.e. every node in $\mathcal{T}_\mathcal{C}$ (except for the root) has a corresponding column in $\mathcal{X}$. In this case lemma 2 directly establishes a recursive definition for the columns of $\mathcal{X}$:

$$\mathcal{X}_v = \begin{cases} \mathbf{e}^N_{\text{doc}(v)} & \text{if } v \text{ is a leaf} \\ \sum_{u \in \text{ch}(v)} \mathcal{X}_u & \text{otherwise.} \end{cases} \tag{4}$$

Here $\mathbf{e}^N_i$ is the i^{th} canonical basis vector for $\mathbb{R}^N$ and $\text{doc}(v)$ indicates the document index leaf v is labelled with. Importantly, (4) shows that the column corresponding to any internal node can be expressed as a simple linear combination of the columns of its children. This basic property lies at the core of our fast multiplication algorithm.

We now show how to apply the reasoning behind (4) to the more general case when $\mathcal{V}$ is an arbitrary subset of V, i.e. a node's children may be partly missing. Define $\mathcal{T}_\mathcal{C}(\mathcal{V}) = (\hat{V}, \hat{E})$, the *restriction* of $\mathcal{T}_\mathcal{C}$ to $\mathcal{V}$, to be a tree with nodes $\hat{V} = \mathcal{V} \cup \{\text{root}(\mathcal{T}_\mathcal{C})\}$. In addition, for any $v \in V \backslash \{\text{root}(\mathcal{T}_\mathcal{C})\}$ let $\text{la}(v, \hat{V}) \in \hat{V}$ be the closest *proper ancestor* of v in $\mathcal{T}_\mathcal{C}$ that is also in $\hat{V}$; since $\text{root}(\mathcal{T}_\mathcal{C}) \in \hat{V}$, this mapping is always well defined. The edge set $\hat{E}$ preserves the ancestor relationships among the nodes in $\hat{V}$: every $v \in \mathcal{V}$ is connected to $\text{la}(v, \hat{V})$ as a child. An inductive argument shows that if $u, v \in \hat{V}$, then u is an ancestor of v in $\mathcal{T}_\mathcal{C}$ *if and only if* u is also an ancestor of v in $\mathcal{T}_\mathcal{C}(\mathcal{V})$.

Associated with $\mathcal{T}_\mathcal{C}(\mathcal{V})$ is a matrix $\Phi \in \mathbb{Z}_+^{N \times |\mathcal{V}|}$ that subsumes the role of leaf document labels. Φ contains a column for every node $v \in \mathcal{V}$ and accounts for all of the leaves in $\mathcal{T}_\mathcal{C}$. When v is a leaf in $\mathcal{T}_\mathcal{C}$ and v is included in $\mathcal{V}$ we set $\Phi_v = \mathbf{e}^N_{\text{doc}(v)}$. Otherwise, v is accounted for in $\Phi_{\text{la}(v, \hat{V})}$, the column pertaining to v's closest ancestor in $\mathcal{V}$. In particular, if $u \in \mathcal{V}$ is *not* a leaf in $\mathcal{T}_\mathcal{C}$, then

$$\Phi_u = \sum_{\substack{v \in \text{leaves}(\mathcal{T}_\mathcal{C}) \backslash \mathcal{V} \\ \text{la}(v, \hat{V}) = u}} \mathbf{e}^N_{\text{doc}(v)}. \tag{5}$$

This bookkeeping allows us to relate the columns of $\mathcal{X}$ when $\mathcal{V}$ is any subset of V:

Theorem 1. *The columns of the node matrix $\mathcal{X}$ for $\mathcal{V} \subseteq V \setminus \{root(\mathcal{T}_\mathcal{C})\}$ are given recursively by*

$$\mathcal{X}_v = \Phi_v + \sum_{u \in ch(v; \hat{E})} \mathcal{X}_u$$

where Φ and $\mathcal{T}_\mathcal{C}(\mathcal{V}) = (\hat{V}, \hat{E})$ are defined above.

This theorem shows that $\mathcal{X}_v$ is a simple linear combination of the columns of its children in $\mathcal{T}_\mathcal{C}(\mathcal{V})$ plus a correction term in Φ. We utilize this structure below to give a fast matrix-vector multiplication algorithm for node matrices.

2.2 Fast Multiplication Algorithm

A simple application of Theorem 1 shows that the matrix-vector product $\mathcal{X}w$ for $w \in \mathbb{R}^{|\mathcal{V}|}$ can be obtained by recursively collecting entries of w into a vector $\beta \in \mathbb{R}^{|\mathcal{V}|}$:

$$\beta_v = w_v + \beta_{\text{p}(v; \hat{E})} \tag{6a}$$

$$\mathcal{X}w = \Phi\beta \tag{6b}$$

Here we use the convention $\beta_{\text{root}(\mathcal{T}_\mathcal{C}(\mathcal{V}))} = 0$. The transposed operation $\mathcal{X}^T y$ for $y \in \mathbb{R}^N$ can also be written recursively by expressing each entry as

$$(\mathcal{X}^T y)_v = \Phi_v^T y + \sum_{u \in \text{ch}(v; \hat{E})} (\mathcal{X}^T y)_u. \tag{7}$$

Equations (6-7) lead to the following theorem, for which we provide a proof sketch:

[4] Note that $\mathcal{V}$ never includes the root node.

Theorem 2. *Let $\mathcal{C}$ be a document corpus of n words and let $\mathcal{X}$ be any node matrix derived from this corpus. Then $\mathcal{X}$ requires $O(n)$ memory to store. Multiplying a vector with $\mathcal{X}$ or $\mathcal{X}^T$ requires $O(n)$ operations.*

Proof. Vector β in equation (6) can be computed in $O(|\mathcal{V}|) \in O(n)$ operations by a top-down traversal that computes each of its entries in constant time. The matrix Φ is a sparse matrix with *at most* one entry per leaf of the suffix tree $\mathcal{T}_\mathcal{C}$, i.e. at most n entries. It follows that the product $\Phi\beta$ requires $O(n)$ operations which proves the theorem for multiplication with $\mathcal{X}$. The transposed case follows similarly by noting that we must compute a matrix-vector product with Φ^T and perform a bottom-up traversal that performs constant time operations for every node in $\mathcal{V}$. □

2.2.1 Efficiency Gains

We use naïve multiplication to mean sparse matrix-vector multiplication in what follows. The supplementary material discusses examples which show that naïve multiplication with the N-gram matrix X can be asymptotically slower than naïve multiplication with $\mathcal{X}$, which in turn can be asymptotically slower than multiplication with our recursive algorithm. These examples establish the following complexity separation result:

Theorem 3. *There exists documents of n words for which*

1. *The all N-grams matrix X requires $\Theta(n^2)$ storage and operations to multiply naïvely.*
2. *The all N-grams node matrix $\mathcal{X}$ requires $\Theta(n\sqrt{n})$ storage and operations to multiply naïvely.*
3. *In all cases recursive multiplication of the node matrix requires $O(n)$ storage and operations.*

2.3 Matrix Data Structure

The fast multiplication algorithm can be performed directly on the suffix tree derived from $\mathcal{C}$, but it is faster to use a dedicated data structure optimized for the algorithm's memory access patterns. The breadth-first multiplication tree (BFMT) stores the topology of $\mathcal{T}_\mathcal{C}(\mathcal{V})$ in the BFF (discussed in section 1.3.2) and the frequency information in Φ as a sparse matrix in a modifed compressed sparse column (csc) format (see the supplementary material) whose columns are ordered according to the order of the BFF. We chose this format because executing equations (6) and (7) requires a simple linear sweep of the BFMT. We expect that the vectors being multiplied can be stored in memory and therefore opted for the speed afforded by the BFF instead of the memory savings of the DFF.

The total number of bits necessary to store the BFMT is given by equation (3) along with the total number of bits necessary to store Φ, which is

$$|\mathcal{V}|\lceil \log_2 U^\Phi \rceil + \text{nz}(\lceil \log_2 M \rceil + \lceil \log_2 N \rceil). \tag{8}$$

Here $U^\Phi = \max_{v\in\mathcal{V}} \|\Phi_v\|_0$ is the largest number of non-zero elements in a column of Φ, nz is the total number of non-zero elements in Φ, and M is the largest entry in Φ. It is easy to verify that $|\mathcal{I}| \leq |\mathcal{V}| \leq \text{nz} \leq n$ and the term involving nz typically dominates the memory requirements.

3 Common Usage Patterns

We now discuss how several common machine learning scenarios can be adapted to use our representation of the node matrix or preferably, to treat multiplication with $\mathcal{X}$ as a black-box routine. The most straightforward use case is to replace the original N-gram matrix with the more succint node matrix. Moreover, mean centering and column normalization can be performed *implictly*, without modifying $\mathcal{X}$, by premultiplying and adding a correction term:

$$\left(\left(\mathcal{X} - \mathbf{1}\mu^T\right)\Sigma\right) w = \mathcal{X}(\Sigma w) - (\mu^T \Sigma w)\mathbf{1}$$

Here μ is a vector of column means and Σ is a diagonal matrix of column normalizing factors. Analogous formulas exist for row centering and normalization.

3.1 Problem Reformulation

A variety of optimization problems can also be reformulated so that they are *equivalent* to using the original N-gram matrix. A simple example of such a conversion comes from using ridge regression to model label $y_i \in \mathbb{R}$ based on the i^{th} row of the N-gram matrix X. We wish to solve

$$\underset{w\in\mathbb{R}^d}{\text{minimize}}\, \frac{1}{2}\|y - Xw\|_2^2 + \frac{\lambda}{2}\|w\|_2^2. \tag{9}$$

It is easy to show that if $\lambda > 0$ and N-grams s, t belong to the same equivalence class, then $w_s = w_t$. We can simulate the effect of these duplicated variables by collecting terms. Let $\mathcal{S}$ be the set of N-grams present in X, $\mathcal{V}$ the set of suffix tree nodes present in X, and define $\mathcal{S}(v) = S(v) \cap \mathcal{S}$ for brevity. For all $v \in \mathcal{V}$ let $z_v = |\mathcal{S}(v)|w_s$ for some $s \in \mathcal{S}(v)$. Then $\sum_{s\in\mathcal{S}(v)} X_s w_s = \mathcal{X}_v z_v$ and $\sum_{s\in\mathcal{S}(v)} w_s^2 = |\mathcal{S}(v)|^{-1} z_v^2$ so problem (9) is equivalent to a *smaller* weighted ridge regression using $\mathcal{X}$:

$$\underset{z\in\mathbb{R}^{|\mathcal{V}|}}{\text{minimize}}\, \frac{1}{2}\|y - \mathcal{X}z\|_2^2 + \frac{\lambda}{2}\sum_{v\in\mathcal{V}} \frac{z_v^2}{|\mathcal{S}(v)|}. \tag{10}$$

Note that this also shows that representations using the N-gram matrix downweight the ridge penalty of each equivalence class in proportion to its size.

We can characterize the set of optimization problems that have an equivalent problem where the N-gram matrix can

be replaced with the node matrix. Define a partition $\mathcal{J}$ of the integer set $\{1, \ldots, d\}$ to be a set of m integral intervals $\zeta_k = \{i, \ldots, j\}$ such that $\bigcup_{k=1}^{m} \zeta_k = \{1, \ldots, d\}$ and $\zeta_k \cap \zeta_j = \emptyset$ if $k \neq j$. A function $f : \mathbb{R}^d \to \mathbb{R}^p$ is *permutation invariant* with respect to $\mathcal{J}$ (abbrev. $\mathcal{J}$-PI) if for all $x \in \mathbb{R}^d$, $f(x) = f(\pi[x])$ where $\pi : \mathbb{R}^d \to \mathbb{R}^d$ is any permutation that only permutes indices within the same interval $\zeta_k \in \mathcal{J}$. For our purposes it is important to note that L_p-norms are $\mathcal{J}$-PI as are affine functions $Ax+b$ whenever columns $A_i = A_j \; \forall i, j \in \zeta_k, \forall \zeta_k \in \mathcal{J}$. It is straightforward to show that if $f, g : \mathbb{R}^d \to \mathbb{R}^p$ are both $\mathcal{J}$-PI and $c : \mathbb{R}^p \to \mathbb{R}^q$ then $f(x) + g(x)$ and $c(f(x))$ are also $\mathcal{J}$-PI.

We prove the following theorem in the supplementary material to connect permutation invariance to optimization:

Theorem 4. *Let $f : \mathbb{R}^d \to \mathbb{R}$ be any convex function that is $\mathcal{J}$-PI where $m = |\mathcal{J}|$. Then there exists a convex function $g : \mathbb{R}^m \to \mathbb{R}$ over m variables such that the problem $min_{x \in \mathbb{R}^d} f(x)$ is equivalent to $min_{z \in \mathbb{R}^m} g(z)$. If z^* is optimal for the second problem, then $x_i = z^*_k \; \forall i \in \zeta_k, \forall \zeta_k \in \mathcal{J}$ is optimal for the first problem.*

This theorem establishes that any convex loss of the form $L(Xw, b)$; e.g. SVM, logisitic regression, least squares; added to any L_p norm, e.g. L_2 ridge or L_1 lasso penalty, can be simplified to an equivalent learning problem that uses the node matrix instead of the N-gram matrix.

3.2 Holding Out Data

Oftentimes the document corpus is organized into T (possibly overlapping) integral sets $\mathcal{Q}_1, \ldots, \mathcal{Q}_T$ indexing the documents. For instance, splitting documents into training and testing sets yields $T = 2$ index sets, and further subdividing the training set for K-fold crossvalidation introduces $2K$ additional sets (indicating the hold out and training data for each split). In this case we are not interested in multiplying all of $\mathcal{X}$, but only the submatrix whose rows' indices are in the given $\mathcal{Q}_i$. This matrix-vector product can be computed by calling the recursive multiplication algorithm with the topology information in $\mathcal{T}_\mathcal{C}(\mathcal{V})$ (derived from the full corpus) and with the submatrix of Φ whose rows' indices are in $\mathcal{Q}_i$. Also note that if only a subset of the documents will ever be used for training, we can ignore any nodes in $\mathcal{T}_\mathcal{C}$ that do not appear in the training set since they (should) be ignored by any learning algorithm; we discuss this further in section 4.2.

4 Preprocessing

We use an intermediary data structure, the depth-first preprocessing tree (DFPT), to output the BFMT. The DFPT is computed from a corpus $\mathcal{C}$ and stores the *minimal* information in $\mathcal{T}_\mathcal{C} = (V, E)$ necessary to produce any BFMT and to prune the nodes in V. It can be computed once and used to store $\mathcal{C}$ in a format that is amenable for arbitrary machine learning tasks. Given a new learning problem the DFPT proceeds in two stages: 1) it identifies useful N-grams in V and calculates relevant column normalizations, and 2) it emits a BFMT tailored to that task. Construction of the DFPT, as well as its processing stages, requires $O(n)$ time and memory with respect to the corpus length n.

As suggested by its name, the DFPT stores the topology of $\mathcal{T}_\mathcal{C}$ in DFF, its leaf-document annotations, and if filtering by N-gram length, the edge label length for each internal node of V. Its processing stages are a sequence of of top-down/bottom-up traversals of $\mathcal{T}_\mathcal{C}$ that are individually more sophisticated than those required by our multiplication algorithm, so we opted for the memory savings afforded by the DFF. Indeed, depth($\mathcal{T}_\mathcal{C}$) is bounded by the length of the longest document in $\mathcal{C}$ while the tree width is bounded by the corpus length; the memory savings of the DFF over the BFF are substantial. Importantly, the traversals stream through the DFPT so it is reasonable to operate on it via external memory, e.g. a hard drive, if memory is limited.

In detail, the DFPT requires $2|I|\lceil \log_2 |\Sigma| \rceil + n\lceil \log_2 N \rceil$ bits to store the topology and leaf-document annotations, where I is the set of internal nodes of V, N the number of documents in $\mathcal{C}$, and Σ the alphabet. For reference storing $\mathcal{C}$ requires $n\lceil \log_2 |\Sigma| \rceil + N\lceil \log_2 n \rceil$ bits and $|I| < n$. An additional $|I|\lceil \log_2 \frac{n}{N} \rceil$ bits are used to store edge labels lengths, but this information is only necessary when pruning by maximum N-gram length.

4.1 Computing the Depth-First Suffix Tree

Computing the DFPT from $\mathcal{C}$ represents the least memory efficient part of our framework as we first compute a suffix array (SA) (Gusfield, 1997) from the text and then convert the SA into the DFPT. The process requires $3n\lceil \log_2 n \rceil + n\lceil \log_2(|\Sigma| + N) \rceil$ bits and $O(n)$ time. We emphasize that our framework is completely modular so the DFPT only needs to be computed once. We leave it as an open problem to determine if a more memory efficient algorithm exists that directly computes the DFPT.

Recalling that each leaf of $\mathcal{T}_\mathcal{C}$ is numbered according to the suffix it corresponds to, the SA is a permutation of the integers $[0, \ldots, n)$ that stores the leaves of $\mathcal{T}_\mathcal{C}$ in a *pre-order depth-first traversal*. We use an SA rather than a suffix tree because the former typically requires 4 times less memory than a suffix tree and can also be constructed in $O(n)$ time and memory. We use the implementation of (Mori, 2015), which requires $m = 3n\lceil \log_2 n \rceil + n\lceil \log_2(|\Sigma| + N) \rceil$ bits to construct the SA, where the second term corresponds to a modified copy of $\mathcal{C}$. This was the most memory efficient linear-time suffix array construction algorithm we could find; asymptotically slower but more memory efficient algorithms may be preferable for DNA sequence data.

The details of how we compute the DFPT from a suffix

array are rather involved and will be discussed in an extended version of this paper. The framework of (Kasai, 2001) is instrumental in our conversion as it allows us to simulate a *post-order depth-first traversal* of $\mathcal{T}_\mathcal{C}$ using the SA. By carefully managing memory and off-loading unused information to external storage, each step of the conversion requires at most $m - n\lceil\log_2 n\rceil$ bits to be stored in main memory at any time. The *total* memory requirements, including storing the DFPT while it is constructed, never exceed the maximum of $m - n\lceil\log_2 n\rceil$ and $2n\lceil\log_2 n\rceil + (n + |I|)\lceil\log_2 N\rceil$ bits; both are less than m.

4.2 Filtering and Normalizations

The first stage of the DFPT's processing determines which nodes in $\mathcal{T}_\mathcal{C}$ should be present in the final BFMT. It also computes any required normalizations, such as the column mean or norm, of the node matrix $\mathcal{X}$ the BFMT represents. We assume that only the internal nodes $I \subset V$ of $\mathcal{T}_\mathcal{C}$ will ever be used; each leaf appears in only a single document and is unlikely to carry useful information. We model the screening process as a sequence of filters that are applied to I: associated with I is a boolean array $b \in \{0,1\}^{|I|}$ where $b_v = 1$ indicates that node $v \in I$ is useful and $b_v = 1\ \forall v \in I$ initially. Each filter takes as input the DFPT and b, and updates b (in place) with its own criteria. All of our filters are memory efficient and only need to store $|I| + O(\text{depth}(\mathcal{T}_\mathcal{C}))$ bits in memory as the BFMT can reasonably be streamed from slower external storage. With the exception of the unique document filter, all of the filters listed below run in $O(n)$ time:

N-gram length: removes nodes whose shortest corresponding N-gram is longer than a given threshold.

Training set: removes nodes that do not appear in any documents designated as the training set.

Unique document frequency: removes nodes that do not appear in at least some number of *distinct* documents. We use an algorithm given in (Paskov, 2015) which runs in $O(n\alpha^{-1}(n))$ time, where α^{-1} is the inverse Ackermann function ($\alpha^{-1}(10^{80}) = 4$) and is essentially linear-time. A $O(n)$ algorithm (Gusfield, 1997) is possible, but it requires complicated pre-processing and an additional $n\lceil\log_2 n\rceil$ bits of memory.

Strong rules: given *mean centered* document labels $y \in \mathbb{R}^N$, removes all nodes v for which $|\mathcal{X}_v^T y| < \lambda$ for a threshold λ. This implements the strong rules of (Tibs., 2010) and can be applied to a subset of the documents $\mathcal{I}_{\text{tr}} \subset \{1, \ldots, N\}$ (e.g. training data) by mean centering only $y_{\mathcal{I}_{\text{tr}}}$ and setting $y_i = 0$ for all $i \notin \mathcal{I}_{\text{tr}}$. Column normalizations are achieved by checking $\eta_v^{-1}|\mathcal{X}_v^T y| < \lambda$, where η_v^{-1} is the normalization for column v. This filter essentially multiplies $\mathcal{X}^T y$ using the DFPT and the normalization can be computed on the fly (see discussion below).

Once we know which nodes will be used in the BFMT we typically require the column mean $\mu = \frac{1}{N}\mathcal{X}^T\mathbf{1}$ and some kind of normalization η_v^{-1} for each column of $\mathcal{X}$. Noting that all entries of $\mathcal{X}$ are non-negative, the L_1-norm of each column is $\eta = \mathcal{X}^T\mathbf{1}$. Each of these quantities is a matrix-vector multiply that we perform using the DFPT. These quantities can be specialized to training data by setting appropriate entries of the $\mathbf{1}$ vector to 0. We can also compute the L_2-norm of each column of $\mathcal{X}$ or the L_1/L_2-norm of each column of $\mathcal{X} - \mathbf{1}\mu^T$, the mean centered node matrix. These normalizations however require $O(N|I|)$ time and $O(N\text{depth}(\mathcal{T}_\mathcal{C}))$ memory; the space savings of the DFF are critical for the memory bound. These running times are tolerable if performed only once, especially on the short and wide trees that tend to occur with natural language.

4.3 Producing the Matrix Structure

The final stage in our pipeline produces the BFMT using the DFPT and filter b. The following lemma follows from the definitions of breadth-first and depth-first traversals and is essential for easy conversion between the two formats:

Lemma 3. *Given a tree $T = (V, E)$, let β be an (ordered) list of the nodes in V in breadth-first order and define δ to be a list of V in depth-first preorder. Define $\beta(d)$ and $\delta(d)$ to be the (sub) lists of β and δ respectively containing only nodes at depth d. Then $\beta(d) = \delta(d)\ \forall d = 1, \ldots, \text{depth}(T)$.*

This lemma states that the breadth-first and depth-first preorder traversals list nodes in the same order if we only consider the nodes of a tree at a specific depth. We thus allocate memory for the BFMT by counting the number of nodes with $b_v = 1$ at each depth in the DFPT. The lemma then allows us to copy the relevant nodes in the DFPT into the BFMT skeleton by maintaining a stack of size depth($\mathcal{T}_\mathcal{C}$) that keeps track of how many nodes have been written to the BFMT at each depth. The depth-first traversal also makes it is easy to determine edges by keeping track of each node's nearest ancestor (in $\mathcal{T}_\mathcal{C}$) that is in the BFMT. The copying process streams through the DFPT and b in a single linear sweep and requires storing the BFMT and $O(\text{depth}(\mathcal{T}_\mathcal{C}))$ bits in memory.

5 Experiments

This section provides benchmarks for our multiplication algorithm and applies it to solve several large-scale sentiment analysis tasks. We implemented our framework in C [5] and compiled it used the GCC compiler version 4.4.7 for an x86-64 architecture. Our reference machine uses an Intel Xeon E5-2687W processor with 8 cores running at 3.1 GHz and has 128 Gb of RAM.

[5] Please contact the first author for source code.

5.1 Memory and Timing Benchmarks

We evaluate our multiplication algorithm on three kinds of data to investigate its performance in a variety of scenarios: short natural language articles, long technical papers, and DNA sequence data. The first is the BeerAdvocate dataset (McAuley, 2013), a corpus of $1,586,088$ beer reviews totalling 1 Gb of plaintext and each consisting of a median of 126 words. The second is a collection of $70,728$ journal articles collected from NCBI (U.S. National Library, 2015) with a median length of 6955 words and totalling 3 Gb of plaintext[6]. Our third dataset is derived from the 1000 Genomes Project (1000 Genomes Project, 2008) and it consists of $6,196,151$ biallelic markers, i.e. binary values, along chromosome 1 for 250 people.

Our preprocessing framework required at most 3.5 times the memory of the original datasets for the natural language data. The third dataset however presents a worst case scenario for our framework and suffix tree/arrays in general. It requires 185 megabytes to store because of its small alphabet size, yet its suffix array requires $n\lceil \log_2 n \rceil$ bits, i.e. 31 times more memory, and several times this amount to compute. While the DFPT ameliorates this memory usage, it still requires 10 times more memory than the original data and total memory usage went up to 18 gigabytes when computing it from the suffix array.

Figure 1 compares the memory requirements of the BFMT to explicitly storing the node and N-gram matrices for all N-grams up to length K that appear in at least 2 documents. We show space requirements for our modified sparse matrix format (MSF) as well as the standard sparse format (SSF), e.g. used in Matlab. The top two graphs correspond to the natural language datasets and have similar patterns: memory usage for the explicit representations rises quickly for up to $K = 7$ and then tapers off as overly long N-grams are unlikely to appear in multiple documents. In all cases the BFMT is superior, requiring approximately 3 times less memory than the MSF and up to 14 times less memory than its floating-point counterpart. While there is some separation between the node matrix and naïve all N-gram matrix, the gap – which is more pronounced in the journal articles – is mitigated by filtering N-grams that do not appear in multiple documents.

The third graph presents a striking difference between the representations: the BFMT requires up to 41 times less memory than the MSF node matrix and over $4,600$ times less memory than the naive N-gram matrix. The floating point counterparts for these matrices accentuate the differences by a factor of 5. Interestingly, the size of the BMFT *decreases* as K increases from 10^3 to 10^4. This occurs because when $K \geq 10^4$, the BFMT behaves as if all N-grams are included so that all entries in the frequency matrix Φ are

[6]This data was graciously made available by the Saccharomyces Genome Database at Stanford.

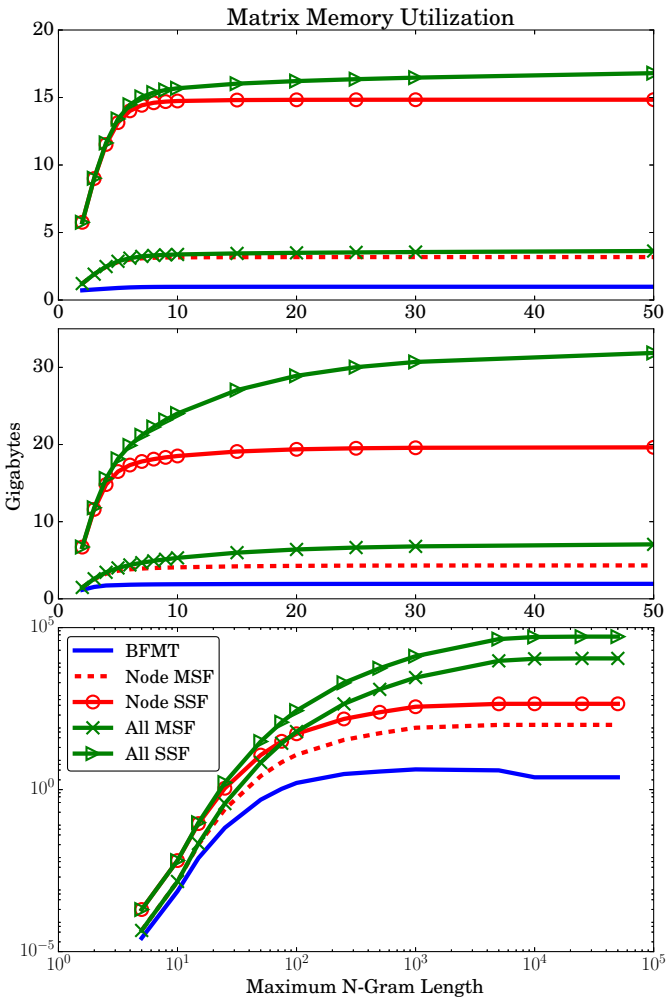

Figure 1: Memory utilization for the BFMT, node, and all N-gram matrices as a function of maximum N-gram length K on the BeerAdvocate data (top), journal data (middle) and 1000 genomes data (bottom).

0 or 1. When $K \approx 10^3$, most of the entries are bounded by 1, but a few large entries exist and force additional bits to be used for *all* non-zero frequencies in Φ.

Next, figure 2 compares the average multiplication time for the BFMT to ordinary sparse multiplication with the node matrix. The top figure presents results for the BeerAdvocate data; we did not include timings for the journal data since they are essentially the same. We were unable to provide timing results for the node matrix on the DNA data because it quickly exceeded our computer's memory. All trials were performed using a single core for fairness of comparison. The difference between the BFMT and the node matrix closely follows the memory requirement differences. This is to be expected as both multiplication routines make a single pass over the data so running time is proportional to the amount of data that must be scanned. We also note that the BFMT running time scales gracefully

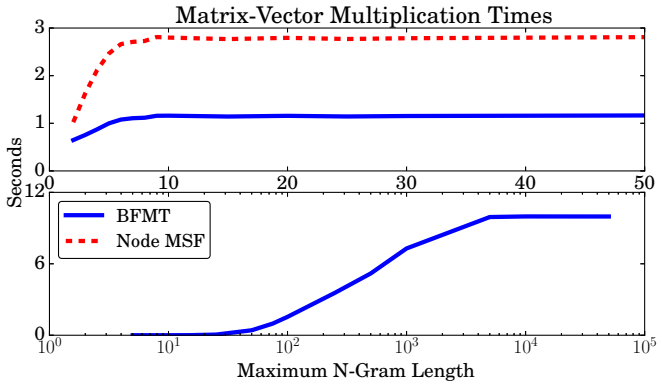

Figure 2: Average time to perform one matrix-vector multiply with the BFMT and node matrices as a function of maximum N-gram length K on the BeerAdvocate data (top) and 1000 Genomes Data (bottom). Node matrix times are missing for the latter because it was impractical to store.

on the DNA data; time increases at a logarithmic rate with respect to K since the x-axis is logarithmic.

5.2 Sentiment Analysis Tasks

We applied our framework to sentiment analysis tasks on three large datasets: the BeerAdavocate dataset, a set of $6,396,350$ music reviews from Amazon (McAuley, 2013) (4.6 Gb of text), and a set of $7,850,072$ movie reviews also from Amazon (McAuley, 2013) (7.4 Gb of text). Each review's sentiment is a value between 1 and 5 (indicating negative or positive) and we tried to predict this sentiment using a ridge regression model on features provided by the node matrix. Each dataset was randomly split into training, validation, and testing sets comprised of $75\%, 12.5\%$, and 12.5% of the total data; all parameters discussed below were selected based on their validation set performance.

We solved the regression by implementing a conjugate gradient solver in C that uses our fast multiplication routine. The ridge parameter λ was tuned on a grid of up to 100 values. We stopped tuning once the validation error increased for 5 consecutive λ values and the procedure typically terminated after trying 60 values. N-grams were pruned by maximum N-gram length and were required to appear in at least 20 distinct documents – we experimented with several document frequency thresholds. We also used the strong rules to select a subset of the features for each λ value and used $\alpha\lambda$ as the threshold; $\alpha = 1$ always gave the best performance. Finally, all columns were mean centered and normalized by their L_2 norm. Our framework computed this normalization in 2.5 minutes for the larger movie dataset. The largest and most time intensive feature set contained 19.4 million features and occured for $K = 5$ on the movie dataset. It took 26 hours to solve for and evaluate 69 lambda values while running on a *single* core. We were able to effectively run all N-gram trials in parallel.

Table 1: Mean Squared Error for Sentiment Analysis

K	Beer	Music	Movies
1	0.286	0.766	0.765
2	0.254	0.481	0.237
3	0.245	0.366	0.140
4	**0.244**	0.333	0.121
5	**0.244**	**0.325**	**0.115**

Table 1 summarizes the mean-squared error of our regression model on the testing set. All three datasets benefit from longer N-grams, but we note that the longer datasets seem to benefit more (size increases from left to right). Confounding this potential effect are peculiarities specific to the tasks and specific to BeerAdvocate versus Amazon reviews (recall that the music and movie reviews both come from the same data source). Nonetheless, it is also possible that larger datasets are better equipped to utilize long N-grams: they provide enough examples to counter the variance incurred from estimating coefficients for long, and therefore relatively infrequent, N-grams. It will be interesting to verify this potential effect with more experiments.

6 Conclusion

This paper shows that learning with long N-grams on large corpora is tractable because of the rich structure in N-gram matrices. We provide a framework that can be used to permanently store a document corpus in a format optimized for machine learning. Given a particular learning task, our framework finds helpful N-grams and emits a data structure that is tailored to the problem and allows for fast matrix-vector multiplication – an operation that lies at the heart of many popular learning algorithms.

Our work suggests a number of extensions. While our multiplication algorithm is single-threaded, the underlying data structure is a tree and the necessary traversals are easy to parallelize. Finding an efficient algorithm to directly compute the DFPT without using a suffix array will also be useful. Finally, our framework provides considerable computational savings for DNA sequence data over naïve representations (up to 4 orders of magnitude). We hope that our algorithm will inspire additional research into long N-gram models for this kind of data.

Acknowledgements

In loving memory of Христо П. Георгиев (Hristo P. Georgiev). Funding provided by the Air Force Office of Scientific Research and the National Science Foundation.

References

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. of Discrete Algorithms*, 2(1):53–86, March 2004.

[2] William Cavnar and John Trenkle. N-gram-based text categorization. In *In Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, pages 161–175, 1994.

[3] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, NY, USA, 1997.

[4] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[5] John Houvardas and Efstathios Stamatatos. N-gram feature selection for authorship identification. In *Proceedings of the 12th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, AIMSA'06, pages 77–86, Berlin, Heidelberg, 2006. Springer-Verlag.

[6] Juhani Kähärä and Harri Lähdesmäki. Evaluating a linear k-mer model for protein-dna interactions using high-throughput selex data. *BMC Bioinformatics*, 14(S-10):S2, 2013.

[7] Toru Kasai, Hiroki Arimura, and Setsuo Arikawa. Efficient substring traversal with suffix arrays.

[8] Julian McAuley and Jure Leskovec. From amateurs to connoisseurs: Modeling the evolution of user expertise through online reviews. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 897–908, Republic and Canton of Geneva, Switzerland, 2013. International World Wide Web Conferences Steering Committee.

[9] Julian McAuley and Jure Leskovec. Hidden factors and hidden topics: understanding rating dimensions with review text. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 165–172. ACM, 2013.

[10] Yuta Mori. sais, https://sites.google.com/site/yuta256/sais, 2015.

[11] Jorge Nocedal and Stephen Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer New York, 2006.

[12] U.S. National Library of Medicine. National center for biotechnology information, 2015.

[13] Neal Parikh and Stephen Boyd. Proximal algorithms. *Foundations and Trends in Optimization*, 2013.

[14] Hristo Paskov, John Mitchell, and Trevor Hastie. Uniqueness counts: A nearly linear-time online algorithm for the multiple common substring problem. In *In preparation*, 2015.

[15] Hristo Paskov, Robert West, John Mitchell, and Trevor Hastie. Compressive feature learning. In *NIPS*, 2013.

[16] 1000 Genomes Project. 1000 genomes project, a deep catalog of human genetic variation., 2008.

[17] Konrad Rieck and Pavel Laskov. Linear-time computation of similarity measures for sequential data. *The Journal of Machine Learning Research*, 9:23–48, 2008.

[18] Choon Hui Teo and S. V. N. Vishwanathan. Fast and space efficient string kernels using suffix arrays. In *In Proceedings, 23rd ICMP*, pages 929–936. ACM Press, 2006.

[19] Robert Tibshirani. Regression shrinkage and selection via the lasso. *J. R. Stat. Soc. B*, 58(1):267–288, 1996.

[20] Robert Tibshirani, Jacob Bien, Jerome Friedman, Trevor Hastie, Noah Simon, Jonathan Taylor, and Ryan J. Tibshirani. Strong rules for discarding predictors in lasso-type problems., 2010.

[21] SVN Vishwanathan and Alexander Johannes Smola. Fast kernels for string and tree matching. *Kernel methods in computational biology*, pages 113–130, 2004.

[22] Sida Wang and Christopher Manning. Baselines and bigrams: Simple, good sentiment and topic classification. In *Proceedings of the ACL*, pages 90–94, 2012.

[23] Frank Wood, Jan Gasthaus, Cédric Archambeau, Lancelot James, and Yee Whye Teh. The sequence memoizer. *Communications of the ACM*, 54(2):91–98, 2011.

[24] Dell Zhang. Extracting key-substring-group features for text classification. In *In Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD 06*, pages 474–483. ACM Press, 2006.