Multiplying N-gram Matrices in Linear Space and Time

Kevin Tan tankevin@stanford.edu
Varun Tandon varunt@stanford.edu
German Enik germans@stanford.edu
Eric Lou erlou@stanford.edu

June 11, 2020

1 Abstract

Many modern machine learning models use matrices to encode the information required to solve a problem; these matrices are often multiplied with vectors many millions of times during the training process, which makes finding efficient algorithms for performing these multiplications a task of great practical interest. Generally speaking, matrix-vector multiplication takes *quadratic* time and space. However, in the case of N-gram matrices (a popular feature representation used in natural language models), there exist techniques to perform this operation in *linear* time and space. In this article, we explore the mechanics and inspiration for these techniques, and, along the way, probe the surprisingly deep connections between N-gram matrices and generalized suffix trees.

2 Matrix-Vector Multiplication in Linear Space and Time

Before we begin, let's address the possibility of matrix-vector multiplication in linear space and time.

2.1 Cursory Analysis

In terms of space, if we're trying to store a 3×3 matrix, shouldn't this take 9 units[1] of memory? Similarly, if we're trying to store a 4×4 matrix, shouldn't this take 16 units of memory? More generally, if we wanted to store an $n \times n$ matrix, shouldn't this take n^2 units (a quadratic amount) of memory?

In terms of time, if we're trying to multiply a 3×3 matrix with a 3×1 vector, don't we need at least 9 multiplications and 6 additions? Similarly, if we wanted to multiply a 4×4 matrix with a 4×1 vector, don't we need at least 16 multiplications and 12 additions? More generally, if we wanted to multiply an $n \times n$ matrix and an $n \times 1$ vector, don't we need at least n^2 (a quadratic number) multiplications and $n(n-1)$ (also a quadratic number) additions?

Are we on a wild goose chase? Or is our analysis somehow incorrect?

Rest assured, our analysis is *not* incorrect, but it *does* assume that we're working with *arbitrary* matrices. It most definitely takes quadratic space to store an *arbitrary* matrix and quadratic time to multiply an *arbitrary* matrix with a vector. But if we restrict ourselves to matrices with a linear number of nonzero entries-a type of *sparse* matrix-then what we want is indeed possible. Let's see why.

2.2 Storing Matrices in Linear Space

If we only have a linear number of nonzero entries, the overwhelming majority of the entries are zero. Where there's a lot of redundancy, there's a lot of potential for compression. In particular, instead of explicitly storing every entry of a matrix M, let's only explicitly store the nonzero ones; the zero entries can be stored *implicitly*. One can imagine implementing this with a hierarchical map $\mathcal{M}$ with two layers of keys.

- **Layer 1:** The first key into $\mathcal{M}$ represents the row of a matrix. That is, $\mathcal{M}[i] = M_i$.
- **Layer 2:** The second key into $\mathcal{M}$ represents the column of a matrix. That is, $\mathcal{M}[i][j] = M_{ij}$.

In particular, we only add the nonzero entries into map $\mathcal{M}$. If the map doesn't have an associated value when the first key is r and the second key is c, then we know that $M_{rc} = 0$. In this manner, we've successfully stored the nonzero entries implicitly without consuming space for them. Because we only have a linear number of nonzero entries by assumption, storing such a matrix only takes linear space.

2.3 Multiplying Matrices in Linear Time

Given the representation of sparse matrices described in the previous section, it's not too difficult to see that we can matrix-vector multiplication in linear time. Specifically, since the zero entries would not have contributed meaningfully to the result anyways, we can simply loop over the entries of $\mathcal{M}$, multiply them with the corresponding element in the vector, and then aggregate the results.

Because we loop over a linear number of nonzero entries, and perform a constant amount of work for each entry, this also takes linear time overall. The conclusion, then, is that if we have a matrix with a linear number of nonzero entries, then we can both store the matrix in linear space and multiply it in linear time. The question then becomes: do N-gram matrices obey this constraint?

[1] For some measure of memory, be it bits or bytes.

3 Basic Terminology

Before we can answer this question, let's define what an N-gram matrix is. Along the way, we pick up a few other pieces of terminology and notation.

3.1 Document and Corpus

Informally, a document is a string and a corpus is a collection of documents. Formally, a document $D = d_1 d_2 \dots d_N$ is a sequence of characters d_i and a corpus $C = \{D_1, D_2, \dots, D_M\}$ is a set of documents D_i. For instance, below are some examples[2] of documents. Collectively, they form a corpus.

$$D_1 = \texttt{THE□ONLY□SUBSTITUTE□FOR□GOOD□MANNERS□IS□FAST□REFLEXES}$$
$$D_2 = \texttt{ARTIFICIAL□INTELLIGENCE□IS□NO□MATCH□FOR□NATURAL□STUPIDITY}$$
$$D_3 = \texttt{TALK□IS□CHEAP□UNTIL□YOU□HIRE□A□LAWYER}$$
$$C = \{D_1, D_2, D_3\}$$

3.2 N-gram and N-gram Set

Informally, an N-gram is a string of N letters[3] and an N-gram set is a collection of N-grams. Formally, an N-gram $S = s_1 s_2 \dots s_n$ is a sequence of characters s_i and an N-gram set $\mathcal{S} = \{S_1, S_2, \dots, S_m\}$ is a set of N-grams S_i. For instance, below are some examples of N-grams. Collectively, them form an N-gram set.

$$S_1 = \texttt{W} \qquad S_4 = \texttt{XE} \qquad S_7 = \texttt{STR}$$
$$S_2 = \texttt{G} \qquad S_5 = \texttt{CH} \qquad S_8 = \texttt{NTE}$$
$$S_3 = \texttt{R} \qquad S_6 = \texttt{ST} \qquad S_9 = \texttt{□IS}$$

$$\mathcal{S} = \{S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9\}$$

3.3 N-gram Matrix

An N-gram matrix X is a data structure that stores the frequencies with which the N-grams in an N-gram set $\mathcal{S}$ appear in a corpus C. For instance, take our corpus from section 3.1 and N-gram set from section 3.2. The corresponding N-gram matrix is shown below. Though arbitrary, it's conventional to have the rows represent the documents and columns represent the N-grams.

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9
D_1	0	1	3	1	0	2	0	0	1
D_2	0	1	3	0	1	1	0	1	1
D_3	1	0	2	0	1	0	0	0	1

Because this notation will be useful later on, let X_S be the column in the N-gram matrix corresponding to the N-gram S. Here are some examples of this notation, so you can get a feel for what it means.

$$X_{S_1} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \qquad X_{S_5} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \qquad X_{S_8} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

[2] In these examples, the □ symbol represents a space.

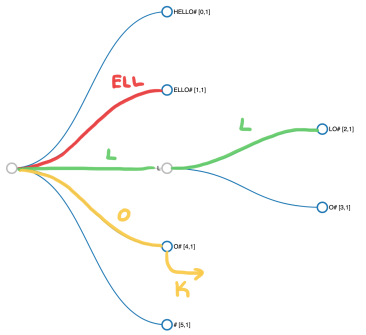
[3] An N-gram could also mean a sequence of N words, phonemes, or syllables. While our analysis rests on the assumption that an N-gram is a sequence of letters, it is by no means limited to it. In particular, the contents of this article can be easily generalized to other interpretations of what an N-gram could mean; one can imagine preprocessing a string, assigning to each word/phoneme/syllable an identifier, and then applying our analysis to those identifiers.

4 Using Suffix Trees to Understand N-gram Matrices

At this point we've established that if we can somehow reformulate an N-gram matrix in terms of a sparse matrix, then we can store it using linear space and multiply it in linear time. This section is about how this is possible. But first, let's see how we can use suffix trees to answer essential questions about N-gram matrices. In the following sections, let S be an N-gram, D be a document, and T be the suffix tree for D.

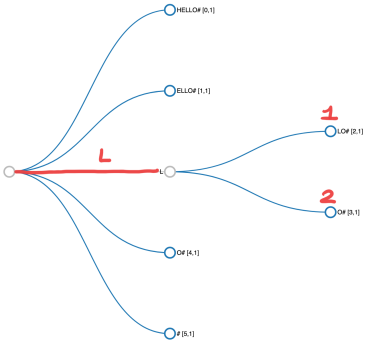
4.1 Does S Appear in D?

First of all, what does it mean for S to *appear* in D? One way to think about this is that S appears in D if S is a substring of D, but this just begs the question: what does it mean for S to be a substring of D? The Fundamental Theorem of Stringology tells us that S is a substring of D if and only if S is a prefix of a suffix of D. While being quite the tongue-twister, this theorem is incredibly useful. Why? Recall that the suffixes of D are formed by concatenating the characters along a root-leaf path in T. So, S is a prefix of a suffix of D only if we don't fall of the tree when using the characters of S to traverse T. This leads to an extremely intuitive algorithm for finding out whether S appears in D.



4.2 How many times does S appear in D?

By the Fundamental Theorem of Stringology, this is equivalent to the question: how many suffixes of D is S a prefix of? Just like before, let's use S to navigate T. The suffixes which S is a prefix of can be constructed by appending to S the characters along the remaining paths to leaf nodes (assuming we stay on the tree, otherwise the answer is trivially zero). Because the remaining number of distinct paths to leaves is exactly equal to the remaining number of leaves, the number of times S appears in D is equal to the number of leaves in the subtree rooted at S[4]. This can be found efficiently with a simple depth-first traversal.



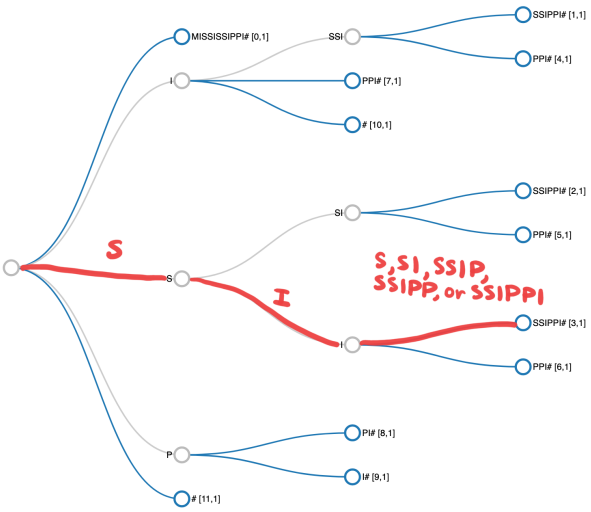
[4]Should processing the letters of S leave you in the middle of an edge, then the subtree is rooted at the next possible node.

5 Developing Useful Abstractions

In machine learning tasks, removing redundant features can yield improvements in memory usage, training time, and model performance. For N-gram matrices, this amounts to removing N-grams that always has the same frequency as some other N-gram. Let's see how we can use T to identify such N-grams in X.

5.1 Equivalence Classes of N-grams and the Node Matrix $\mathcal{X}$

To make our discussion concrete, let $D = \texttt{MISSISSIPPI}$. Note that the N-grams `SIS`, `SISI`, `SISSIP`, `SISSIPP`, and `SISSIPPI` have exactly the same frequencies; this is no coincidence. From the suffix tree, it's clear that if you use any of these N-grams to traverse T, you end up at (or right before) the same node. Because this is the case, these N-grams will all have the same frequencies, as we demonstrated in section 4.2. In general, there are certain *equivalence classes* of N-grams corresponding to the nodes of the suffix tree, the only exception being the root. Whenever you have equivalence classes, it suffices to only consider a single representative element from each class.



The upshot is that some N-grams can be safely eliminated from $\mathcal{S}$ without impacting model performance while decreasing memory usage and training time, which removes potentially a large number of columns from X. We call the resulting matrix the *node matrix* and give it the symbol $\mathcal{X}$. As we will see in the next section, pruning these redundant N-grams actually unlocks further speedups by revealing interrelationships between columns of the $\mathcal{X}$.

5.2 From a Mechanical to an Operational Description

With the analysis we performed in section 5, we can revisit the question we asked ourselves in section 4.2. Previously, we had a *mechanical* description of the answer: start at the root of the suffix tree T, make your way down towards the leaves using the characters of S, and perform a DFS to determine the number of leaf nodes that are still reachable. Now that we know there exist equivalence classes of N-grams and that it's really these equivalence classes that constitute meaningful features, we can develop an *operational* description: the frequency with which S appears in D is equal to the number of leaves in the subtree rooted at the representative node for the equivalence class of S.

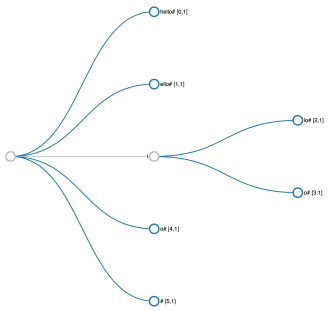
6 Properties of the Node Matrix $\mathcal{X}$

Armed with the operational description from section 5.2, we'll see that even after eliminating useless N-grams, there's *still* redundant information in our N-gram matrix. In particular, the frequency with which node appears is equal to the frequencies with which the node's children appear. Why? Because the number of leaves in the subtree rooted at a node is equal to the sum of the number of leaves in the subtrees rooted at the node's children, and our operational description translates these statements about leaves in subtrees into equivalent statements about frequencies of appearance.

6.1 A Simple Example

Consider the simple example when $D =$ `HELLO`. Below are the N-gram matrix and suffix tree for this document. For simplicity, we've included a representative element from every equivalence class. We will see later how to relax this assumption. As we can see, the column for `L` is the sum of the columns for `LLO` and `LO`. In the suffix tree, the nodes for `LLO` and `LO` are the children of the node for `L`.

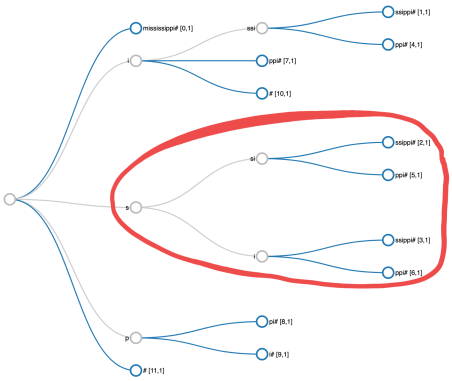
	HELLO	ELLO	L	LLO	LO	O
D	1	1	2	1	1	1



6.2 A Complex Example

Now for a more complex example, let $D =$ `MISSISSIPPI`, where we've only included in the N-gram matrix columns in the subtree marked in red for clarity. This time, notice that the column for `S` is the sum of the columns for `SSI` and `SI`. If we look at the suffix tree, the nodes associated with `SSI` and `SI` are children of the node associated with `S`. There are other examples of these linear dependencies for this document, and at multiple depths in the tree. Can you spot them all?

	S	SSI	SSISSIPPI	SSIPPI	SI	SISSIPPI	SIPPI
D	4	2	1	1	2	1	1



7 Using Generalized Suffix Trees to Understand N-gram Matrices

While we didn't explicitly mention it, we actually made two simplifying assumptions thus far in our analysis. First, we limited ourselves to a single document, which made X and $\mathcal{X}$ simple row vectors. Second, we constrained ourselves to the case where $\mathcal{S}$ includes every equivalence class of N-grams, which made the linear dependencies in X and $\mathcal{X}$ easier to reason about. In this section, we relax the first assumption. Relaxing the second has been left as an exercise for the adventurous reader.

7.1 Generalized Suffix Trees

Very little of our analysis actually changes when we consider multiple documents; the most notable difference is that we now need to build a *generalized* suffix tree. Because such data structures are not the focus of this article, we won't dwell on their intricacies. It suffices to say that this is done by taking a corpus, appending unique sentinel characters to each document so we can differentiate the suffixes of one document from those of another, and constructing a Patricia trie over the suffixes of these slightly modified documents. Let's revisit some of the previous questions we asked ourselves to see how the generalized suffix tree will help us.

7.2 Does S appear in D_i?

Contrast this with what we asked ourselves in section 4.1. With multiple documents, we need to specify the specific document D_i that we're interested in determining whether or not S appears in. Previously, our algorithm consisted of walking down T using the characters of S and reporting whether or not we fall off the tree. What this algorithm was doing was telling us whether S appears in *any* of the documents in our corpus C; this worked because C was just the singleton containing D. Generalizing this to multiple documents requires an additional step. We perform the same tree traversal as before, reporting that S does not appear in D_i if we fall off, but, if we don't, we need to perform a DFS to determine if any of the leaves in the subtree belong to D_i. If so, then S appears in D_i. Otherwise, it does not (it appears in some other D_j where $i \neq j$).

7.3 How many times does S appear in D_i?

The procedure for answering this question is almost exactly the same as that for answering the question in the previous section. The only difference is that we're not interested in whether or not any leaves in the subtree satisfy the property that they belong to D_i, but, rather, how many leaves satisfy this property. The practical consequence of this is that instead of stopping at the first leaf that belongs to D_i in the subtree we reach after navigating T with S, we actually need to explore all of the leaves in the subtree and count the number of leaves that belong to D_i.

7.4 Linear Dependencies of Columns

An interesting and important observation is that the columns of the N-gram matrix are still interrelated when we generalized to multiple documents. The reason why this is the case is because these interrelationships hold for each individual row in the N-gram matrix and, because the rows do not interact, these same relationships still hold when we consider all of the rows simultaneously.

8 Multiplying N-gram Matrices in Linear Space and Time

Alas! Our journey has come to an end; everything is in place to discuss how we can multiply N-gram matrices in linear space and time. To be pedantic, this technique doesn't apply to arbitrary N-gram matrices but specifically to node matrices (N-gram matrices without redundant columns). This isn't really a limitation, however, because, as mentioned in section 5.1, this is almost always what we want.

8.1 Computing $\mathcal{X}w$ in Linear Space and Time

We saw in section 7.4 that if p was a node in T and $c_1, \ldots, c_n$ were its children, then $\mathcal{X}_p = \sum_i \mathcal{X}_{c_i}$. This is an interesting observation, but what does it have to do with efficient matrix-vector multiplication? Well, let's assume for the moment that p and $c_1, \ldots, c_n$ were the only nodes in our generalized suffix tree. This means that our node matrix might[5] look something like

$$\mathcal{X} = \begin{pmatrix} \mathcal{X}_{c_1} & \ldots & \mathcal{X}_{c_n} & \mathcal{X}_p \end{pmatrix}.$$

8.1.1 Reformulating the Operation

Let's see what insights we might derive by actually multiplying the corresponding matrix with a vector w, recalling that matrix-vector multiplication can be thought of as a linear combination of the matrix columns.

$$\begin{aligned} \mathcal{X}w &= w_{c_1}\mathcal{X}_{c_1} + \cdots + w_{c_n}\mathcal{X}_{c_n} + w_p\mathcal{X}_p \\ &= w_{c_1}\mathcal{X}_{c_1} + \cdots + w_{c_n}\mathcal{X}_{c_n} + w_p\sum_i \mathcal{X}_{c_i} \\ &= (w_{c_1} + w_p)\mathcal{X}_{c_1} + \cdots + (w_{c_n} + w_p)\mathcal{X}_{c_n} \end{aligned}$$

Notice that the only columns that remain are the columns that correspond to the children of p. In fact, we can reformulate $\mathcal{X}w$ as $\Phi\beta$ where Φ and β are defined in the following manner.

$$\begin{aligned} \Phi &= \begin{pmatrix} \mathcal{X}_{c_1} & \ldots & \mathcal{X}_{c_n} & 0 \end{pmatrix} \\ \beta &= \begin{pmatrix} w_{c_1} + w_p & \ldots & w_{c_n} + w_p & 0 \end{pmatrix}^T \end{aligned}$$

This insight—the ability to reformulate our original operation in terms of a different matrix and vector—is a general one and is not limited to the specific example considered above. The interpretation of this result is that we don't *actually* need to multiply any of the columns corresponding to an internal node p of T; we can *simulate* this by passing the number we would've needed to multiply $\mathcal{X}_p$ by—w_p—down to p's children.

8.1.2 Analyzing the Space and Time Complexity

In general, the only nonzero columns in matrix Φ are going to be the ones that correspond to the leaves in T. Because there are a linear number of leaves in a suffix tree, there are a linear number of nonzero columns in Φ. Since leaves correspond to suffixes and each suffix can only belong to one document (due to the unique sentinels we appended to each document during the construction of T), each of these columns has exactly one nonzero entry. This means that Φ satisfies the sparsity condition described in section 2, which allows Φ to being stored in linear space and multiplied in linear time. Constructing β can also be done in linear time by performing a simple top down traversal of the suffix tree, which has a linear number of nodes.

[5] Any permutation of the columns would be an equally valid node matrix.

8.2 Computing $\mathcal{X}^T y$ in Linear Space and Time

Depending on the context, it may be necessary to multiply the transpose of the node matrix instead. This can also be done in linear time. To see why this is the case, let's reuse the setup from section 8.1.

$$\mathcal{X}^T = \begin{pmatrix} \mathcal{X}_{c_1} & \dots & \mathcal{X}_{c_n} & \mathcal{X}_p \end{pmatrix}^T$$

8.2.1 Reformulating the Operation

Let's see what insights we might derive by actually multiplying $\mathcal{X}^T$ with a vector y, recalling that matrix-vector multiplication can be thought of as taking the dot product of the matrix rows with the vector y.

$$\begin{aligned} \mathcal{X}^T y &= \begin{pmatrix} \mathcal{X}_{c_1}^T y & \dots & \mathcal{X}_{c_n}^T y & \mathcal{X}_p^T y \end{pmatrix}^T \\ &= \begin{pmatrix} \mathcal{X}_{c_1}^T y & \dots & \mathcal{X}_{c_n}^T y & (\sum_i \mathcal{X}_{c_i})^T y \end{pmatrix}^T \\ &= \begin{pmatrix} \mathcal{X}_{c_1}^T y & \dots & \mathcal{X}_{c_n}^T y & \sum_i \mathcal{X}_{c_i}^T y \end{pmatrix}^T \end{aligned}$$

This insight suggests that we can simply compute the dot products for the children and simply add them together to get the value in the output vector corresponding to the parent. In contrast to the previous section where the tree traversal was top-down and performed before the multiplication takes place, we now have to traverse the tree bottom-up and do this as the multiplication is taking place.

8.2.2 Analyzing the Space and Time Complexity

We've already address why Φ takes a linear amount of space to store. All that remains is analyzing why this operation takes linear time to compute. It takes a constant amount of time to multiply such vectors. Since we need to perform a linear number of such multiplications because there is a linear number of leaves, this takes linear time. Filling out the other entries of the result vector with a bottom up traversal of the suffix tree also takes linear time because it involves just adding a constant number of things.

9 Conclusion

The N-gram matrix, a critical component of many popular machine learning models in the field of natural language processing, can be represented in linear space and multiplied in linear time. This is done by first recognizing and removing redundant N-grams which leaves us with a node matrix $\mathcal{X}$, and then resolving the linear dependencies between the remaining columns which leaves us with a leaf label matrix Φ which has a linear number of nonzero entries; these kinds of matrices can be represented in linear space and multiplied in linear time. Multiplying $\mathcal{X}$ with a vector w requires transforming w into β via a top down traversal of the corresponding generalized suffix tree, propagating weights downward. Multiplying $\mathcal{X}^T$ with a vector y requires a bottom up traversal of the corresponding generalized suffix tree, propagating intermediate results upward. None of these incredible improvements would have been possible without having recognized and explored the intimate relationship between N-gram matrices and generalized suffix trees.

10 Creative Component

We made a website that contains everything in this paper, a lot of interactive visualizations for better understanding, and bench-marking analysis of our implementation of section 8.1. Here's the link: https://varuntandon.github.io/FastMatrixMultiplicationSuffixTrees/