EFFICIENT CALCULATION OF POLYNOMIAL FEATURES ON SPARSE MATRICES

Nystrom, Andrew
awnystrom@gmail.com [*†]

Hughes, John
jfh@cs.brown.edu [†]

ABSTRACT

We provide an algorithm for polynomial feature expansion that operates directly on and produces a compressed sparse row matrix without any densification. For a vector of dimension D, density d, and degree k the algorithm has time complexity $O(d^kD^k)$ where k is the polynomial order, which is an improvement by a factor d^k over the standard method.

1 INTRODUCTION

Polynomial feature expansion has long been used in statistics to approximate nonlinear functions Gergonne (1974); Smith (1918). The compressed sparse row (CSR) matrix format is a widely-used data structure to hold design matrices for statistics and machine learning applications. However, polynomial expansions cannot be performed directly on sparse CSR matrices, or any sparse matrix format for that matter, without intermediate densification steps. This densification not only adds extra overhead, but causes combinations of features that have a product of zero to be computed, then put back into a sparse format, which is futile.

We provide an algorithm that allows CSR matrices to be the input of a polynomial feature expansion without any densification. The algorithm leverages the CSR format to only compute products of features that result in a nonzero value. This exploits the sparsity of the data to achieve an improved time complexity of $O(d^kD^k)$ on each vector of the matrix where k is the degree of the expansion, D is the dimensionality, and d is the density. The standard algorithm has time complexity $O(D^k)$. Since $0 \leq d \leq 1$, our algorithm is a significant improvement. While the algorithm we lay out uses CSR matrices, it could be modified to operate on other sparse formats.

2 PRELIMINARIES

A matrix $\boldsymbol{A}$ is uppercase and bold. A row vector $\boldsymbol{a}_i$ has a subscript and is the i^{th} row of $\boldsymbol{A}$, whereas $\boldsymbol{a}$, without a subscript, is a vector not necessarily related to $\boldsymbol{A}$.

A compressed sparse row (CSR) matrix representation of $\boldsymbol{A}$ consists of three vectors: $\boldsymbol{c}$, $\boldsymbol{d}$, and $\boldsymbol{p}$. $\boldsymbol{c}$ and $\boldsymbol{d}$ hold the column indices and data values, respectively, of all nonzero elements of $\boldsymbol{A}$. The values of $\boldsymbol{p}$ index $\boldsymbol{c}$ and $\boldsymbol{d}$ and is itself indexed by rows of $\boldsymbol{A}$ such that the nonzero columns of $\boldsymbol{a}_i$ are $\boldsymbol{c}_{\boldsymbol{p}_i:\boldsymbol{p}_{i+1}}$ and the nonzero data elements are $\boldsymbol{d}_{\boldsymbol{p}_i:\boldsymbol{p}_{i+1}}$. Since only nonzero elements of each row are held, the dimensionality must also be stored since it cannot be implicitly gotten from the length of $\boldsymbol{c}$.

Scalers, vectors, and matrices are often referenced with the superscript k. This is not to be interpreted as an exponent, but to indicate that it is the analogous aspect of that which procedes it, but in its polynomial expansion form. For example, $\boldsymbol{c}^k$ is the vector that holds columns for nonzero values in $\boldsymbol{A}$'s polynomial feature expansion CSR representation.

For simplicity, all polynomial expansions are assumed to be of the second degree. We do provide an algorithm for third degree expansions, and derive the big O time complexity of the general case. We have also developed an algorithm for second and third degree interaction features (combinations without repetition), which can be found in the implementation.

[*]Now at Google
[†]The authors contributed equally important and fundamental aspects of this work.

3 MOTIVATION

In this section, an algorithm for computing polynomial feature expansions on dense matrices will be given. Next, the algorithm will be modified slightly to operate on a CSR matrix in order to expose its infeasibility. We will show how the algorithm would be feasible with an added component and derive that component in the following section.

3.1 DENSE EXPANSION ALGORITHM

A natural way to calculate polynomial features for a matrix $\boldsymbol{A}$ is to walk down its rows and, for each row, take products of all k combinations of elements. In order to determine in which column $\boldsymbol{A}^k_i$ products of elements in $\boldsymbol{A}_i$ belong, a simple counter can be set to zero for each row of bmA and incremented efter each polynomial feature is generated. This counter gives the column of bmA^k_i into which each expansion feature belongs.

SECOND ORDER DENSE POLYNOMIAL EXPANSION ALGORITHM($\boldsymbol{A}$)

```
1  $N$ = row count of $\boldsymbol{A}$
2  $D$ = column count of $\boldsymbol{A}$
3  $\boldsymbol{A}^k$ = empty $N \times \binom{D}{2}$ matrix
4  for $i = 0$ to $N - 1$
5      $c_p = 0$
6      for $j_1 = 0$ to $D - 1$
7          for $j_2 = j_1$ to $D - 1$
8              $\boldsymbol{A}^k_{ic_p} = \boldsymbol{A}_{ij_1} \cdot \boldsymbol{A}_{ij_2}$
9              $c_p = c_p + 1$
```

3.2 IMPERFECT CSR EXPANSION ALGORITHM

Now consider how this algorithm might be modified to accept a CSR matrix. Instead of walking directly down rows of bmA, we will walk down sections of $\boldsymbol{c}$ and $\boldsymbol{d}$ partitioned by $\boldsymbol{p}$, and instead of inserting polynomial features into $\boldsymbol{A}^k$, we will insert column numbers into $\boldsymbol{c}^k$ and data elements into $\boldsymbol{d}^k$.

Incomplete Second Order CSR Polynomial Expansion Algorithm($\boldsymbol{A}$)

```
1   $N$ = row count of $\boldsymbol{A}$
2   $\boldsymbol{p}^k$ = vector of size $N + 1$
3   $\boldsymbol{p}^k_0 = 0$
4   $nnz^k = 0$
5   for $i = 0$ to $N - 1$
6       $i_{start} = \boldsymbol{p}_i$
7       $i_{stop} = \boldsymbol{p}_{i+1}$
8       $\boldsymbol{c}_i = \boldsymbol{c}_{i_{start}:i_{stop}}$
9       $nnz^k_i = \binom{|\boldsymbol{c}_i|}{2}$
10      $nnz^k = nnz^k + nnz^k_i$
11      $\boldsymbol{p}^k_{i+1} = \boldsymbol{p}^k_i + nnz^k_i$

    // Build up the elements of $\boldsymbol{p}^k$, $\boldsymbol{c}^k$, and $\boldsymbol{d}^k$
12  $\boldsymbol{p}^k$ = vector of size $N + 1$
13  $\boldsymbol{c}^k$ = vector of size $nnz^k$
14  $\boldsymbol{d}^k$ = vector of size $nnz^k$
15  $n = 0$
16  for $i = 0$ to $N - 1$
17      $i_{start} = \boldsymbol{p}_i$
18      $i_{stop} = \boldsymbol{p}_{i+1}$
19      $\boldsymbol{c}_i = \boldsymbol{c}_{i_{start}:i_{stop}}$
20      $\boldsymbol{d}_i = \boldsymbol{d}_{i_{start}:i_{stop}}$
21      for $c_1 = 0$ to $|\boldsymbol{c}_i| - 1$
22          for $c_2 = c_1$ to $|\boldsymbol{c}_i| - 1$
23              $\boldsymbol{d}^k_n = \boldsymbol{d}_{c_0} \cdot \boldsymbol{d}_{c_1}$
24              $\boldsymbol{c}^k_n = ?$
25              $n = n + 1$
```

The crux of the problem can be found on line 24. Given the arbitrary columns involved in a polynomial feature of $\boldsymbol{A}_i$, there is no way to determine what the corresponding column of $\boldsymbol{A}^k_i$. We cannot simply reset a counter for each row as we did in the dense algorithm. This is because only columns corresponding with nonzero values are stored. Any time a column is implicitly skipped that would have held a zero value, the counter would err.

In order to achieve a general algorithm, we require a mapping from columns of $\boldsymbol{A}$ a column of $\boldsymbol{A}^k$. If there are D columns of $\boldsymbol{A}$ and $\binom{D}{k}$ columns of $\boldsymbol{A}^k$, this can be accomplished by a bijective mapping of the following form:

$$(j_0, j_1, \ldots, j_{k-1}) \rightarrowtail\!\!\!\!\twoheadrightarrow p_{j_0 j_1 \ldots i_{k-1}} \in \{0, 1, \ldots, \binom{D}{k} - 1\} \tag{1}$$

such that $0 \leq j_0 \leq j_1 \leq \cdots \leq j_{k-1} < D$ where $(j_0, j_1, \ldots, j_{k-1})$ are elements of $\boldsymbol{c}$ and $p_{j_0 j_1 \ldots i_{k-1}}$ is an element of $\boldsymbol{c}^k$.

4 Construction of Mapping

Within this section, i, j, and k denote column indices. For the second degree case, we seek a map from matrix indices (i, j) (with $0 \leq i < j < D$) to numbers $f(i, j)$ with $0 \leq f(i, j) < \frac{D(D-1)}{2}$, one that follows the pattern indicated by

$$\begin{bmatrix} x & 0 & 1 & 3 \\ x & x & 2 & 4 \\ x & x & x & 5 \\ x & x & x & x \end{bmatrix} \tag{2}$$

where the entry in row i, column j, displays the value $f(i, j)$. We let $T_2(n) = \frac{1}{2}n(n+1)$ be the nth triangular number; then in Equation 2, column j (for $j > 0$) contains entries with $T_2(j-1) \leq$

$e < T_2(j)$; the entry in the ith row is just $i + T_2(j-1)$. Thus we have $f(i,j) = i + T_2(j-1) = \frac{1}{2}(2i + j^2 - j)$. For instance, in column $j = 2$ in our example (the *third* column), the entry in row $i = 1$ is $i + T_2(j-1) = 1 + 1 = 2$.

With one-based indexing in both the domain and codomain, the formula above becomes $f_1(i,j) = \frac{1}{2}(2i + j^2 - 3j + 2)$.

For *polynomial* features, we seek a similar map g, one that also handles the case $i = j$. In this case, a similar analysis yields $g(i,j) = i + T_2(j) = \frac{1}{2}(2i + j^2 + j + 1)$.

To handle *three-way interactions*, we need to map triples of indices in a 3-index array to a flat list, and similarly for higher-order interactions. For this, we'll need the tetrahedral numbers $T_3(n) = \sum_{i=1}^{n} T_2(n) = \frac{1}{6}(n^3 + 3n^2 + 2n)$.

For three indices, i, j, k, with $0 \leq i < j < k < D$, we have a similar recurrence. Calling the mapping h, we have

$$h(i,j,k) = i + T_2(j-1) + T_3(k-2); \tag{3}$$

if we define $T_1(i) = i$, then this has the very regular form

$$h(i,j,k) = T_1(i) + T_2(j-1) + T_3(k-2); \tag{4}$$

and from this the generalization to higher dimensions is straightforward. The formulas for "higher triangular numbers", i.e., those defined by

$$T_k(n) = \sum_{i=1}^{n} T_{k-1}(n) \tag{5}$$

for $k > 1$ can be determined inductively.

The explicit formula for 3-way interactions, with zero-based indexing, is

$$h(i,j,k) = 1 + (i-1) + \frac{(j-1)j}{2} + \tag{6}$$

$$\frac{(k-2)^3 + 3(k-2)^2 + 2(k-2)}{6}. \tag{7}$$

5 Final CSR Expansion Algorithm

With the mapping from columns of $\boldsymbol{A}$ to a column of $\boldsymbol{A}^k$, we can now write the final form of the innermost loop of the algorithm from 3.2. Let the mapping for $k = 2$ be denoted h^2. Then the innermost loop becomes:

```
for $c_2 = c_1$ to $|\boldsymbol{c}_i| - 1$
    $j_0 = \boldsymbol{c}_{c_0}$
    $j_1 = \boldsymbol{c}_{c_1}$
    $c_p = PolyCol^2(j_0, j_1)$
    $\boldsymbol{d}_n^k = \boldsymbol{d}_{c_0} \cdot \boldsymbol{d}_{c_1}$
    $\boldsymbol{c}_n^k = c_p$
    $n = n + 1$
```

The algorithm can be generalized to higher degrees by simply adding more nested loops, using higher order mappings, modifying the output dimensionality, and adjusting the counting of nonzero polynomial features in line 9.

6 Time Complexity

6.1 Analytical

Calculating k-degree polynomial features via our method for a vector of dimensionality D and density d requires $\binom{dD}{k}$ (with repetition) products. The complexity of the algorithm, for fixed $k \ll$

dD, is therefore

$$\begin{aligned} O\left(\left(\!\binom{dD+k-1}{k}\!\right)\right) &= O\left(\frac{(dD+k-1)!}{k!(dD-1)!}\right) && (8) \\ &= O\left(\frac{(dD+k-1)(dD+k-2)\ldots(dD)}{k!}\right) && (9) \\ &= O\left((dD+k-1)(dD+k-2)\ldots(dD)\right) \text{ for } k \ll dD && (10) \\ &= O\left(d^k D^k\right) && (11) \end{aligned}$$

6.2 Empirical

To demonstrate how our algorithm scales with the density of a matrix, we compare it to the traditional polynomial expansion algorithm in the popular machine library scikit-learn Pedregosa et al. (2011) in the task of generating second degree polynomial expansions. Matrices of size 100×5000 were randomly generated with densities of 0.2, 0.4, 0.6, 0.8, and 1.0. Thirty matrices of each density were randomly generated, and the mean times (gray) of each algorithm were plotted. The red or blue width around the mean marks the third standard deviation from the mean. The time to densify the input to the standard algorithm was not counted.

The standard algorithm's runtime stays constant no matter the density of the matrix. This is because it does not avoid products that result in zero, but simply multiplies all second order combinations of features. Our algorithm scales quadratically with respect to the density. If the task were third degree expansions rather than second, the plot would show cubic scaling.

The fact that our algorithm is approximately 6.5 times faster than the scikit-learn algorithm on 100×5000 matrices that are entirely dense is likely a language implementation difference. What matters is that the time of our algorithm increases quadratically with respect to the density in accordance with the big O analysis.

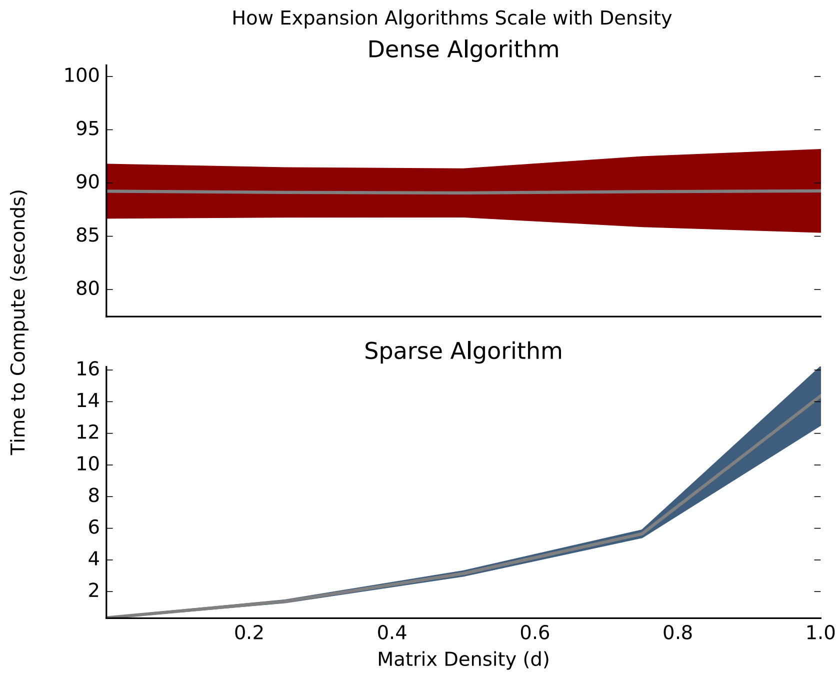

Figure 1: This plot shows how our algorithm (bottom) scales with the density of a matrix compared to a traditional polynomial feature expansion method (top). The task was a second degree expansion, which is why the time of our algorithm scales quadratically with the density.

7 Conclusion

We have developed an algorithm for performing polynomial feature expansions on CSR matrices that scales polynomially with respect to the density of the matrix. While the areas within machine learning this work touches are not en vogue, they are workhorses of industry. This improvement could therefore spare the burning of much fossil fuel.

References

JD Gergonne. The application of the method of least squares to the interpolation of sequences. *Historia Mathematica*, 1(4):439–447, 1974.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Kirstine Smith. On the standard deviations of adjusted and interpolated values of an observed polynomial function and its constants and the guidance they give towards a proper choice of the distribution of observations. *Biometrika*, 12(1/2):1–85, 1918.