Efficient Calculation of Polynomial and Interaction Features on Sparse Matrices

Andrew Nystrom
awnystrom@gmail.com

John Hughes
jfh@cs.brown.edu

Abstract

We give an algorithm to efficiently calculate second degree polynomial or interaction features for a sparse matrix. The algorithm has average time complexity that decreases quadratically with respect to the density of the matrix, which is a large improvement over the naive approach. It also allows for the matrix to remain in sparse form, reducing memory demands. We apply the method to several real datasets and give both analytical and empirical comparisons with the naive approach. The work also gives a generalizable method for creating algorithms for higher orders of polynomials.

1 Introduction

When performing a modeling task, one is often faced with a non-linearly separable problem, or is attempting to predict a target that is a non-linear combination of the variables at one's disposal. There are several general classes of methods for dealing with these situations: kernel methods, high variance models, and input transformations.

Of the possible types of input transformations one might apply, a widely used method is the polynomial transformation, which is defined as all products of all combinations with replacement of k components of a D dimensional row vector $\vec{x}$ (assuming row-wise instance representation), where k is the degree of the polynomial. In the case of $k = 2$, the following set of polynomial features are augmented to each $\vec{x}_i$:

$$\{x_a \cdot x_b : a, b \in \{0, 1, ..., D-1\} \land a \leq b\}$$

which results in $\binom{D+1}{D-1} = \frac{D^2+D}{2}$ additional features, so the generation of these features for a matrix of size $N \times D$ has a time complexity of $O(ND^2)$. In the general case of degree k, there are $\binom{D+k-1}{D-1}$ polynomial features and the time complexity is $O(ND^k)$.

To show the types of nonlinearity polynomial features are capable of imbuing a linear model with, we trained logistic regression models on four inseparable datasets and plot their decision boundaries, as seen in figure (1) on page 2. We compare these decision boundaries to those learnt by more complex methods, namely a support vector machine with a radial basis function kernel[1] and gradient boosted decision trees[2]. As can be seen, polynomial features allow a simple linear classifier is capable of drawing decision boundaries that in the form of arbitrary parabolas, hyperbolas, and elipses.

Beyond classification, polynomial regression [**?**] is a staple regression method. We provide a demonstration of the power polynomial features can lend to a regression problem by fitting two ordinary least squares models to $y = sin(x) + \mathcal{N}(0, 0.1), x \in [0, \pi]$. The first model (the blue curve) was given only x as its input. The second model (the green curve) was given x and second degree polynomial features derived from x (namely x^2). The curves the models learned can be seen in figure 2.

While polynomial features are a well known and used concept in machine learning (e.g. [**?**, **?**, **?**]) and work has been done to improve their utilization (e.g. [**?**, **?**]), this is the first work that we know of that exploits matrix sparcity to improve the speed with which they are calculated and allows them to be calculated for a sparse matrix.

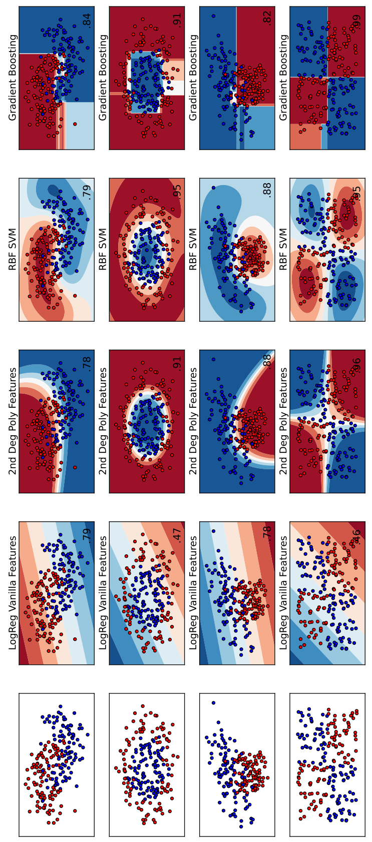

Figure 1: Decision boundaries for four classifier types on four different datasets. The darker the color, the more certain the decision. The plain dataset is shown in the first column. Each other column is a type of classifier applied to that dataset. The classifier types are logistic regression, logistic regression with second degree polynomial features, RBF SVM, and gradient boosting with decision trees. The accuracy of each model is shown in the bottom right corner of that model's subplot.

2 Sparse Polynomial Features

If the vectors of a feature matrix are sparse, many of these products of features will be zero, and are therefore unnecessary to calculate if the matrix is stored in a sparse matrix data structure. If the density (the fraction of nonzero elements) of the matrix is d where $0 \leq d \leq 1$, the number of nonzero polynomial features is

[1]`http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html`

[2]`https://github.com/dmlc/xgboost/`

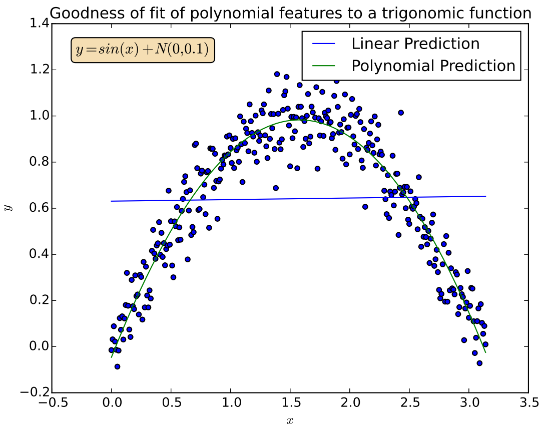

Figure 2: Two linear regression models fit to $y = sin(x) + \mathcal{N}(0, 0.1), x \in [0, \pi]$. The blue line's model was given only x, while the green line's model was given x and second degree polynomial features.

$$\binom{dD + 2 - 1}{dD - 1} = \frac{d^2D^2 + dD}{2}$$

and if only they were calculated, the time complexity of generating polynomial features for a matrix of size $N \times D$ is $O(Nd^2D^2)$, quadratically lower than calculating them via brute force.

The question then is how to devise an algorithm to calculate polynomial features for a vector $\vec{x}$ so that only the nonzero elements of $\vec{x}$ are considered during the calculation process. Two main components are needed: (1) the ability to quickly access only the nonzero elements of $\vec{x}$ and (2) The ability to know which polynomial feature column the product of features with indices a and b should be stored in.

The first of these necessary components is obtained simply by using the appropriate data structure to store the sparse matrix (e.g. storing it in compressed sparse row form). The second component is not only less readily available, but its necessity is less obvious. To make this need more clear, consider the algorithm for the brute force calculation of polynomial features:

```
DENSE POLYNOMIAL FEATURES($A$)
    $N$ = row count of $A$
    $D$ = column count of $A$
    $B$ = Matrix of size $N \times \frac{D^2+D}{2}$
    for $row$ in $A$
        $k = 0$
        for $i = 0$ to $D$
            for $j = i$ to $D$
                $r$ = index of $row$
                $B[r,k] = row[i] \cdot row[j]$
                $k = k + 1$
```

Notice that, for an individual row, the location of a polynomial feature ($row[i] \cdot row[j]$) can be determined by a simple counter k that is incremented each pass through the inner loop. This cannot be done if we only calculate products between nonzero elements of $\vec{x}$, because all that will be known is the nonzero element indices - many iterations will be skipped (hence the improved computational complexity).

What is therefore needed is a mapping between row-column index pairs of $\vec{x}$ and the space $0, 1, ..., \frac{D^2+D}{2} - 1$ (the output size of second degree polynomial features when $D = |\vec{x}|$). The ordering of the mapping is irrelevant so long as its input

$$I := \{(a, b) : a, b \in \{0, 1, ..., D - 1\} \wedge a \leq b\}$$

is bijective with its output

$$O := \{0, 1, ..., \frac{D^2 + D}{2} - 1\}$$

Interaction features are a way of capturing correlations between features in a machine learning setting. A feature vector $\vec{x}$ of dimensionality D has second degree interaction features $\{x_i \cdot x_j : i, j \in \{0, 1, ..., D-1\} \wedge i < j\}$, so a D dimensional vector has $\binom{D}{2} = \frac{D^2-D}{2}$ second degree interaction features. A naive approach to calculating these features is to simply iterate through the combinations of the column indices. For a sparse vector, many of the resulting interaction features would be zero, and could therefore be ignored. This work describes a method to efficiently calculate second degree interaction features for a sparse matrix that has time and space complexities that decrease quadratically with the density of the input matrix with respect to the naive approach.

3 Approach

Let the list of nonzero columns for a given row $\vec{x}$ be denoted by N_{zc}. The nonzero second degree interaction features are simply the products of all combinations of two elements whose columns are in N_{zc}. However, to properly place an interaction feature into the correct column, a mapping from the column index pairs of N_{zc} into the columns of the interaction matrix is needed. The mapping is from pairs (i, j), where i and j are in $0, 1, \ldots, D-1$, and $i < j$, to $0, 1, 2, ..., \frac{D^2-D}{2} - 1$, Such a mapping essentially consists of mapping the indices of entries in the upper triangle of a matrix to indices in a flat list. We now describe the construction of such a mapping.

3.1 Mapping construction

We seek a map from matrix indices (i, j) (with $i < j$ and $0 \le i < D$) to numbers $f(i, j)$ with $0 \le f(i, j) < \frac{D(D-1)}{2}$, one that follows the pattern indicated by

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

where the entry in row i, column j, displays the value $f(i, j)$.

To simplify slightly, we introduce a notation for the nth triangular number,

$$T_2(n) = \frac{n(n+1)}{2} \tag{2}$$

The subscript 2 indicates that these are triangles in two dimensions; we'll use $T_3(n)$ to indicate the nth tetrahedral number, and so on for higher dimensions.

Observe that in Equation 1, each entry in column j (for $j > 0$) lies in the range

$$T_2(j-1) \le e < T_2(j). \tag{3}$$

And in fact, the entry in the ith row of that column is just $i + T_2(j-1)$. Thus we have

$$\begin{aligned} f(i, j) &= i + T_2(j-1) && (4) \\ &= i + \frac{(j-1)j}{2} && (5) \\ &= \frac{2i + j^2 - j}{2}. && (6) \end{aligned}$$

For instance, in column $j = 2$ in our example (the *third* column), the entries range from 1 to 2, while $T_2(j-1) = T_2(1) = 1$ and $T_2(j) = T_2(2) = 3$, and the entry in column $j = 2$, row $i = 1$ is $i + T_2(j-1) = 1 + 1 = 2$.

3.1.1 Other indices

With one-based indexing in both the domain and codomain, the formula above becomes

$$\begin{aligned} f_1(i,j) &= 1 + f(i-1, j-1) && (7)\\ &= 1 + f(i-1, j-1) && (8)\\ &= \frac{2 + 2(i-1) + (j-1)^2 - (j-1)}{2} && (9)\\ &= \frac{2i + j^2 - 3j + 2}{2} && (10) \end{aligned}$$

3.1.2 Polynomial Features

For polynommial features, we seek a map from matrix indices (i,j) (with $i \leq j$ and $0 \leq i < D$) to numbers $g(i,j)$ with $0 \leq f(i,j) < \frac{D(D+1)}{2}$, one that follows the pattern indicated by

$$\begin{bmatrix} 0 & 1 & 3 & 6 \\ x & 2 & 4 & 7 \\ x & x & 5 & 8 \\ x & x & x & 9 \end{bmatrix} \qquad (11)$$

i.e., essentially the same task as before, except that the diagonal is included. One can regard all but the last column of entries in Equation 11 as corresponding to the entries in Equation 1, but shifted to the left. Thus the formula for $g(i,j)$ is simply the formula for f, shifted by 1, i.e.,

$$\begin{aligned} g(i,j) &= f(i, j+1) && (12)\\ &= \frac{2i + (j+1)^2 - (j+1)}{2} && (13)\\ &= \frac{2i + j^2 + j + 1)}{2}. && (14) \end{aligned}$$

Alternatively, we can write this as

$$g(i,j) = i + T_2(j), \qquad (15)$$

and get the same result.

3.1.3 Higher dimensions

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

$$h(i,j,k) = i + T_2(j-1) + T_3(k-2); \qquad (16)$$

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

$$h(i,j,k) = T_1(i) + T_2(j-1) + T_3(k-2); \qquad (17)$$

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

$$T_k(n) = \sum_{i=1}^{n} T_{k-1}(n) \qquad (18)$$

for $k > 1$ can be determined inductively. For $k = 3$, the result is

$$T_3(n) = \sum_{i=1}^{n} T_2(n) \tag{19}$$
$$= \frac{n^3 + 3n^2 + 2n}{6}, \tag{20}$$

so that the formula for 3-way interactions, with zero-based indexing, becomes

$$h(i, j, k) = 1 + (i - 1) + \frac{(j - 1)j}{2} + \tag{21}$$
$$\frac{(k - 2)^3 + 3(k - 2)^2 + 2(k - 2)}{6}. \tag{22}$$

3.1.4 Higher-dimension polynomial features

For the case where we include the diagonal in higher dimensions, we must shift j by 1, k by 2, and so on, and the formula becomes

$$\ell(i, j, k) = T_1(i) + T_2(j) + T_3(k), \tag{23}$$

with analogous formulas for higher degree polynomial interactions.

3.2 Inversion

We not only want to be able to compute the "flat index" $f(i, j)$ corresponding to a particular index pair (i, j), we'd like, given a value $N = f(i, j)$, to be able to determine i and j. We know, given N, that $f(0, j) \leq N \leq f(j, j - 1)$, i.e., that

$$\frac{j^2 - 3j + 2}{2} \leq N \leq \frac{2(j - 1) + j^2 - 3j + 2}{2} \tag{24}$$
$$j^2 - 3j + 2 \leq 2N \leq j^2 - j. \tag{25}$$

If we find the (real number) value j_* for which $j_*^2 - j_* = 2N$, then we know that $j = \lfloor j_* \rfloor$. Finding j_* amounts to solving a quadratic, whence

$$j_* = \frac{1 \pm \sqrt{1 + 8N}}{2}. \tag{26}$$

Clearly the larger value,

$$j_* = \frac{1 + \sqrt{1 + 8N}}{2}, \tag{27}$$

is the one we seek; we set $j = \lfloor j_* \rfloor$, and compute $i = N - \frac{j^2 - j}{2}$. In practice, repeatedly computing square roots may be annoying; instead, when we compute $N = f(i, j)$, we can record the values of i and j along with the interaction features, making inversion a simple matter of looking up these values.

Similarly, for polynomial features, to invert

$$g(i, j) = \frac{2i + j^2 + j + 1)}{2} \tag{28}$$

we know that if $N = g(i, j)$, then $g(0, j) \leq N \leq g(j, j)$, which tells us that

$$2N \leq j^2 + 3j + 1, \text{ so} \tag{29}$$
$$0 \leq j^2 + 3j + (1 - N^2) \tag{30}$$

. Solving for j when this last inequality is an equality, we get

$$j_* = \frac{-3 \pm \sqrt{5 - N^2}}{2}; \tag{31}$$

Once again, we set $j = \lfloor j_* \rfloor$, and compute $i = N - \frac{j^2 + j + 1}{2}$.

Inverting the higher-order formulas, which involve cubics, quartics, etc., is infeasible, but storing the inverse map as described above may be practical, especially for sparse feature sets.

4 The Algorithm

In section X we showed why a mapping from the space X onto the space X is necessary for calculating polynomial features for a sparse matrix, then derived such a mapping.

We now have the necessary components for an algorithm to efficiently calculate polynomial features for a sparse matrix. The matrix needs to be stored in a form that allows for its nonzero column indices, e.g. in sparse row form, and we will use the mapping derived in X to determine which column in the output space the polynomial feature between two input columns is mapping onto. Combining these ideas yields the following algorithm:

```
SPARSE POLYNOMIAL FEATURES($A$)
    PolyMap$(a, b) = \frac{2Da-a^2+2b-3a-2}{2} + a + 1$
    $N$ = row count of $A$
    $D$ = column count of $A$
    $B$ = empty sparse $N \times \frac{D^2+D}{2}$ matrix
    for $row$ in $A$
        $N_{zc}$ = nonzero columns of $row$
        for $i = 0$ to $|N_{zc}|$
            for $j = i$ to $|N_{zc}|$
                $k$ = PolyMap$(N_{zc}[i], N_{zc}[j])$
                $r$ = index of $row$
                $B[r, k] = row[N_{zc}[i]] \cdot row[N_{zc}[j]]$
```

Using the mapping for interaction features and changing the bounds in the loop conditions and the size of the output matrix, the algorithm can be modified to give second order interaction features:

```
SPARSE INTERACTION FEATURES($A$)
    InterMap$(a, b) = \frac{2Da-a^2+2b-3a-2}{2}$
    $N$ = row count of $A$
    $D$ = column count of $A$
    $B$ = empty sparse $N \times \frac{D^2-D}{2}$ matrix
    for $row$ in $A$
        $N_{zc}$ = nonzero columns of $row$
        for $i = 0$ to $|N_{zc}| - 1$
            for $j = i + 1$ to $|N_{zc}|$
                $k$ = InterMap$(N_{zc}[i], N_{zc}[j])$
                $r$ = index of $row$
                $B[r, k] = row[N_{zc}[i]] \cdot row[N_{zc}[j]]$
```

where N_{zc} are the columns of nonzero elements of row *row*.

5 Analysis

5.1 Analytical

We assume the input matrix A is in sparse row form. The steps prior to the first loop are constant operations. The outer loop will be executed N times (once for each row). For each row, the innermost loop will be executed and the inner loop will be $\frac{|N_{zc}|^2+|N_{zc}|}{2}$ times. If the density is uniformly distributed across all rows, $E[|N_{zc}|] = dD$, but this cannot be assumed. In the worst case, then, the innermost loop will each be executed $\frac{D^2+D}{2}$ times, but on average, $\frac{d^2D^2+dD}{2}$ times. The operations in the innermost loop are constant. The time complexity is therefore $O(ND^2)$ and $\Theta(Nd^2D^2)$.

In machine learning, an input matrix often has near uniform density across its rows. For example, if the matrix is sparse due to one-of-m encoding (also known as one-hot vectors), each row vector will have the same number of nonzero components.

5.2 Empirical

To assess the performance of the algorithm on data of varied density, we compare its speed to the PolynomialFeatures class of the popular machine learning scikit-learn [?]. Uniformly random matrices of size $100 \times D$ were generated for $d = 0.1$, 0.2, 0.3, 0.4, and 0.5, and $D = 1$, 101, 201, 301, 401, and 501. Each point was resampled five times and averaged to decrease variance. Since the scikit-learn version of the algorithm does not exploit sparsity, we assume the method's time is invariate with respect the density of the matrix, but we gave it the lowest density of each dimensionality to be fair. The results are shown in figure (3) on page 8.

As can be seen, the sparse method starts out slower than scikit-learn for each density. This is either due to the overhead required by the sparse method (e.g. utilizing the mapping function), or is due to implementation differences. The sparse method eventually becomes faster than scikit-learn for each density. Note the excellent performance of $d = 0.2$. This level of density is not uncommon when working with sparse matrices.

The above benchmark was done on synthetic data. To more realistically determine the performance of the algorithm, we applied it to two real world datasets: 20newsgroups[3] and connect-

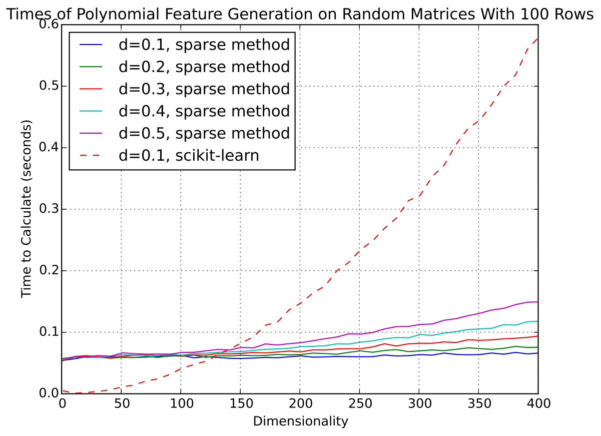

Figure 3: This plot shows how long the sparse polynomial method took to run on matrices with 1000 rows of various dimensionalities and densities and compared to the scikit-learn PolynomialFeatures class on various dimensionalities.

4, which was obtained from the UCI Machine Learning Repository [?][4]. We compare the time and space required by the sparse method and scikit-learn in table (5.2).

Notice that scikit-learn was unable to calculate polynomial features for the 20newsgroups matrix; a memory error was encountered. The sparse method, however, succeeded in 109 seconds. The resulting matrix had nearly 8.5 billion features. While most machine learning libraries would struggle with such a high dimensionality, some, such as the system described in [?] (which is available as part of the Vowpal Wabbit package[5]), would likely be capable of learning a model from such data.

[3]`http://scikit-learn.org/stable/datasets/twenty_newsgroups.html`

[4]`https://archive.ics.uci.edu/ml/datasets/Connect-4`

[5]`https://github.com/JohnLangford/vowpal_wabbit/wiki`

dataset	20newsgroups	connect-4
instances	11,314	67,557
features	130107	126
polynomial features	8,463,980,778	8,127
density	0.12	0.11
dense space (MB)	Memory Error	4191
sparse space (MB)	5333	735
dense time (s)	Memory Error	26
sparse time (s)	109	44

Table 1: Time and space comparison of second degree polynomial feature calculation between the sparse method and scikit-learn.

The connect-4 dataset had its polynomial features computed faster via the dense method than the sparse. This is likely because the dimensionality is sufficiently low for the overhead of the sparse method to not have overtaken the dense method in terms of speed. However, the sparse method took less than a fifth of the memory and took less than twice the speed, so using the sparse method during parameter optimization in a multi-core setting would drastically increase parameter search throughput.

6 Summary & Future Work

In this work we have given a method of calculating second degree polynomial features on sparse matrices with time complexity that quadratically decreases with respect to the density of the matrix.

This work also gives a general pattern for calculating polynomial features of arbitrary degrees: instead of finding a mapping from the 2-tuples of indices onto the space $\{0, 1, ..., \frac{D^2+D}{2} - 1\}$, construct a mapping from k-tuples of indices onto the space $\{0, 1, ..., \binom{D+k-1}{D-1} - 1\}$, where k is the degree of the polynomial. In terms of the upper triangular matrix analogy, this corresponds with mapping the upper k-simplex of a $(D, D, ..., D)$-tensor onto the space $\{0, 1, ..., \binom{D+k-1}{D-1} - 1\}$. For the average case of dD nonzero elements per row, there will be $\binom{dD+k-1}{dD-1}$ polynomial features, so the average time complexity for an $N \times D$ matrix is $\Theta(Nd^kD^k)$, which means that its time complexity polynomially decreases with respect to the density compared to the dense algorithm (which has $O(ND^k)$ and $\Theta(ND^k)$).