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 $\vec{x}_i$:

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

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 help visualize what second degree polynomial features are capable of in a machine learning paradigm, we trained a linear classifier (logistic regression) on four different datasets and show their decision boundaries (figure (1) on page 2) with and without polynomial features, along with their accuracy on a holdout set. We

compare these to an RBF SVM via scikit-learn[1] and gradient boosted decision trees via the xgboost package[2].

To demonstrate the benefit of polynomial features in regression, we 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.

With polynomial features added, a simple linear classifier is capable of drawing decision boundaries that take on the form of arbitrary parabolas, hyperbolas, and elipses. Beyond classification, polynomial regression [3] is a staple regression method. While polynomial features are a well known and used concept in machine learning (e.g. [4,6,9]) and work has been done to improve their utilization (e.g. [2,7]), 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.

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

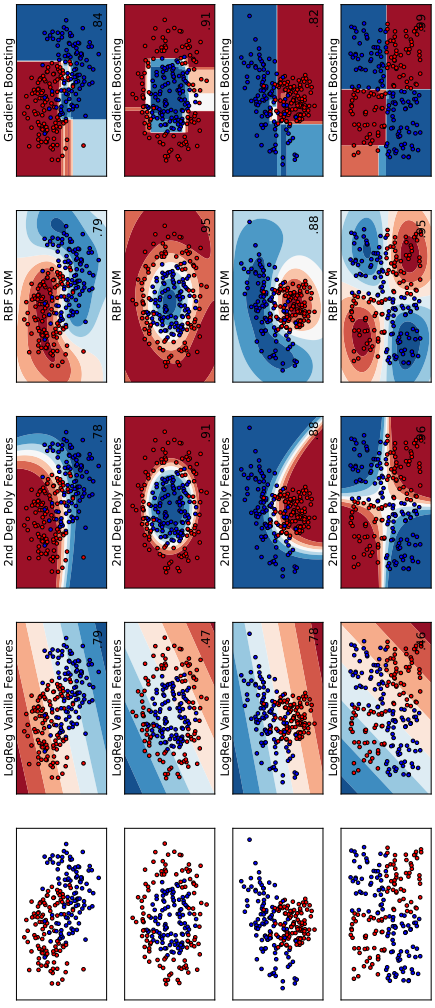

Figure 1: Decision boundaries for four classifier types on four different datasets. The darker the color, the more certain the decision. The plain dataset is shown in the first column. Each other column is a type of classifier applied to that dataset. The classifier types are logistic regression, logistic 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.

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

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

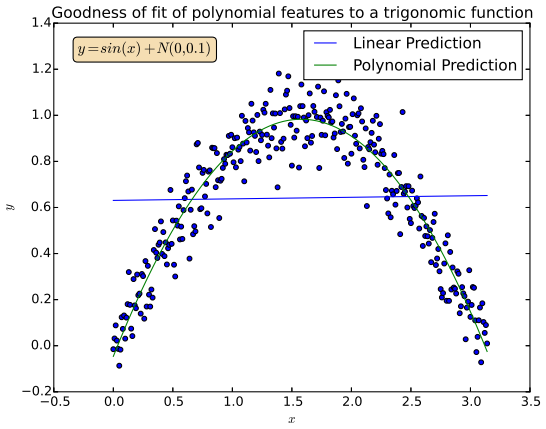

Figure 2: Two linear regression models fit to $y = sin(x) + \mathcal{N}(0, 0.1), x \in [0, \pi]$. The blue line's model was given only x, while the green line's model was given x and second degree polynomial features.

number of nonzero polynomial features is

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

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 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

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

To construct this mapping, consider a matrix of size $D \times D$ where cell (a, b) represents the polynomial feature between indices a and b for $a, b \in I$ for vector $\vec{x}$.

$$\begin{array}{c} \\ x_0 \\ x_1 \\ . \\ . \\ . \\ x_{D-1} \end{array} \begin{array}{c} \begin{array}{ccccccc} x_0 & x_1 & . & . & . & x_{D-1} \end{array} \\ \begin{pmatrix} 0 & 1 & . & . & . & D-1 \\ - & D & . & . & . & 2D-2 \\ . & . & . & & & . \\ . & & . & . & & . \\ . & & & . & . & . \\ - & . & . & . & - & \frac{D^2+D}{2} - 1 \end{pmatrix} \end{array}$$

Again, any bijective mapping $I \mapsto O$ would suffice, but this particular mapping could be considered one of the more natural options as it maps I onto O where both I and O are sorted in ascending order. As can be seen, the required mapping can be viewed as mapping the upper triangular matrix (including the diagonal), whose indicies are I, onto O. The construction of this mapping will now be given.

INSERT JOHN'S PROOF HERE

The mapping can be modified slightly to yield interaction features instead of polynomial features by noting that the section of the $D \times D$ mapping matrix that will yield interaction features differs only in that its diagonal is excluded. The mapping is therefore gotten by taking the polinomial mapping and subtracting $i + 1$.

3 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)

$\text{InterMap}(a, b) = \frac{2Da - a^2 + 2b - 3a - 2}{2}$
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** $|N_{zc}|$
      $k = \text{InterMap}(N_{zc}[i], N_{zc}[j])$
      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*.

4 Analysis

4.1 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, $\frac{d^2 D^2 + dD}{2}$ 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.

4.2 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 [8]. 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 6.

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

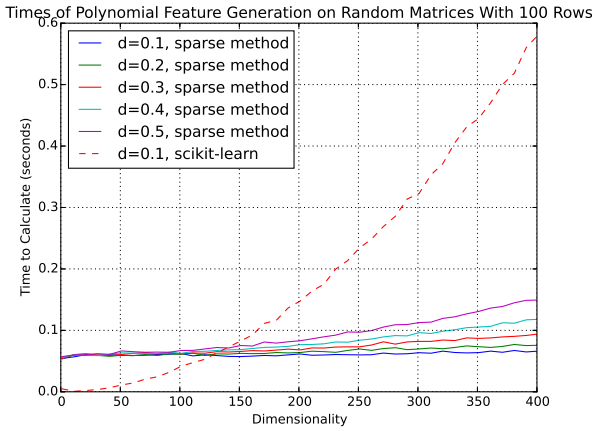

Figure 3: This plot shows how long the sparse polynomial method took to run on matrices with 1000 rows of various dimensionalities and densities and compared to the scikit-learn PolynomialFeatures class on various dimensionalities.

data. To more realistically determine the performance of the algorithm, we applied it to two real world datasets: 20newsgroups[3] and connect-4, which was obtained from the UCI Machine Learning Repository [5][4]. We compare the time and space required by the sparse method and scikit-learn in table (4.2).

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 [1] (which is available as part of the Vowpal Wabbit package[5]), would likely be capable of learning a model from such data.

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.

5 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

[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`

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 $\Theta(ND^k)$).

References

[1] Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *The Journal of Machine Learning Research*, 15(1):1111–1133, 2014.

[2] Ling Huang, Jinzhu Jia, Bin Yu, Byung-Gon Chun, Petros Maniatis, and Mayur Naik. Predicting execution time of computer programs using sparse polynomial regression. In *Advances in Neural Information Processing Systems*, pages 883–891, 2010.

[3] Bent Jorgensen. *Theory of Linear Models*. Chapman & Hall, 1993.

[4] George Konidaris and Andre S Barreto. Skill discovery in continuous reinforcement learning domains using skill chaining. In *Advances in Neural Information Processing Systems*, pages 1015–1023, 2009.

[5] M. Lichman. UCI machine learning repository, 2013.

[6] Paul Pavlidis, Jason Weston, Jinsong Cai, and William Stafford Noble. Learning gene functional classifications from multiple data types. *Journal of computational biology*, 9(2):401–411, 2002.

[7] Aleksandar Pečkov, Sašo Džeroski, and Ljupčo Todorovski. *A minimal description length scheme for polynomial regression*. Springer, 2008.

[8] 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.

[9] Simon Wiesler, Markus Nussbaum-Thom, Georg Heigold, R Schluter, and Hermann Ney. Investigations on features for log-linear acoustic models in continuous speech recognition. In *Automatic Speech Recognition & Understanding, 2009. ASRU 2009. IEEE Workshop on*, pages 52–57. IEEE, 2009.