A New Polynomial Function Approximation Algorithm

Paul E. Utgoff
Jun Qian

Technical Report 99-20
January 28, 1999

Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Telephone: (413) 545-4843
Net: utgoff@cs.umass.edu

Contents

Abstract

We present a new algorithm GMG for building a polynomial approximation of a real-valued function. The method is able to find sparse or dense polynomials of high degree and dimensionality. This is accomplished by a novel organization of the space of monomial terms. The algorithm solves approximation tasks that were previously intractable.

1 Introduction

A longstanding problem in function approximation is to find an efficient method for constructing high degree polynomials in a large number of variables. The problem stems from the exponential growth in the number of possible monomial terms as the number of variables or the degree of the polynomial rises. Under the assumption that most of the high degree terms will not be needed, how can one find the important terms quickly? We propose such a method and demonstrate its advantages.

The problem to solve is as follows. Given a set of N training points $(\mathbf{x}, f(\mathbf{x}))$, find a polynomial in $\mathbf{x}$ that is highly accurate for the training points and for other points that could be drawn from the same distribution. A point $\mathbf{x}$ is represented as a vector of d components in R^d. The target value of the target function f to be approximated is specified as $f(\mathbf{x})$. We assume that N is as large as necessary to sample the function adequately. We do not have any knowledge of the analytic form of f, but we hope to be able to approximate it well by a polynomial in the d variables. The approximation algorithm must find a good approximation $\hat{f}$ in the form of a polynomial. We would like the approximation to exhibit a low mean-squared error (MSE), i.e. $\frac{1}{N}\sum_{i=1}^{N}(\hat{f}(\mathbf{x}_i) - f(\mathbf{x}_i))^2$.

The desire to be able to infer a function from data has a very long history in Mathematics, dating back to Gauss and Newton. The problem is of immense practical value because one can hope to gain insight into a process that one can measure, by attempting to model the data, and then by studying the model.

For functions of many variables or high degree, it is not practical to generate all the possible monomials up to that degree. For d variables, the number of monomials of exactly degree h is:

$$\binom{h+d-1}{h}$$

because it is a d-part composition of h (Reingold, Nievergelt & Deo, 1977). The number of possible monomials is the sum of these exact counts from 0 up to the maximum degree of a monomial, giving:

$$\sum_{i=0}^{h}\binom{i+d-1}{i}.$$

Table 1 shows the total number of possible monomials of degree 1 or higher up to the indicated maximum degree, for the indicated number of variables. One can see that generating all possible monomials is intractable for all but small problems.

Table 1. Number of Possible Monomials

Vars	deg 1	deg 2	deg 3	deg 4	deg 5	deg 6	deg 7	deg 8	deg 9	deg 10
10	11	66	286	1001	3003	8008	19448	43758	92378	184756
9	10	55	220	715	2002	5005	11440	24310	48620	92378
8	9	45	165	495	1287	3003	6435	12870	24310	43758
7	8	36	120	330	792	1716	3432	6435	11440	19448
6	7	28	84	210	462	924	1716	3003	5005	8008
5	6	21	56	126	252	462	792	1287	2002	3003
4	5	15	35	70	126	210	330	495	715	1001
3	4	10	20	35	56	84	120	165	220	286
2	3	6	10	15	21	28	36	45	55	66
1	2	3	4	5	6	7	8	9	10	11

2 The GMG Algorithm

Figure 1 shows the top level of a new algorithm GMG (Greedy Monomial Grabber) for finding a polynomial approximation $\hat{f}$ of the target function f. The main loop appends each currently best monomial to the approximation. The GMG algorithm is efficient to the extent that it generates only those monomials that are needed in the approximation. Unneeded monomials do not derail the process because a coefficient of 0 has the effect of discarding a monomial. However, creating and discarding is less efficient than not creating at all, and we would like to be efficient. To optimize the coefficients, the algorithm computes, in a standard way, a linear regression that minimizes the MSE.

```
$I \leftarrow$ set of $N$ training points
$\hat{f} \leftarrow 0$
open $\leftarrow$ node with 0 degree monomial
while ( MSE($\hat{f}$,$I$) $> \varepsilon$)
   { m $\leftarrow$ best_monomial($\hat{f}$,$I$,open)
     $\hat{f} \leftarrow \hat{f}$ || '+ m'
     open $\leftarrow$ open || successors(m)
     optimize_coefficients($\hat{f}$,$I$)
     update_MSEs($\hat{f}$,$I$,open)
   }
return $\hat{f}$
```

Figure 1. Top Level Algorithm

The best monomial to append next is taken from the head of the open list, using as a metric the MSE of the approximation that would include the candidate monomial. One can compute quite readily the MSE for the N instances that would result if the candidate monomial were to be appended to the approximation, and its coefficients optimized. Candidate monomials with lower MSE are considered better. Upon appending the best monomial, the successors of the monomial are appended to the open list. The coefficients for the current polynomial approximation (including the new monomial just appended), and all the nodes on the open list, are reoptimized, and the MSE of every node on the open list is recomputed. Based on these recomputed MSE values, the order of the nodes on the open list may change. The GMG algorithm continues in this manner until the approximation achieves sufficiently low error, which assumes that such an MSE can be achieved. The algorithm has no closed list; a monomial is either in the approximation, or on the open list, or is yet to be generated.

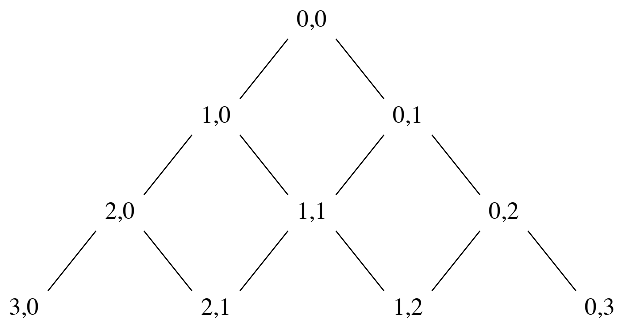

Figure 2. One-Rule Monomial Space

The shape of a search space is determined not only by the metric that is applied to each state/point, but also by the organization of the space, as determined by the order in which states/points are generated. Consider a space, which we call ‘One-Rule Monomial Space’, that is ordered by increasing degree of the monomials. Figure 2 shows a part of this monomial space for a hypothetical problem of two variables. Note that for our purposes, every monomial space contains the possible monomials without coefficients, which are determined separately. To save superscript and subscript clutter, we indicate each monomial solely by a vector of its exponents.

This simple approach (MSE metric, monomial space organization, greedy appending) has several advantages, but two serious problems that we now address. The first is due to the problem of bias in variables of odd versus even power. For example, a variable raised to an even power curves upward for negative values, while a variable raised to an odd power curves downward for negatives. If a variable raised to an odd power is a better class of fits than that variable raised to an even power, then this organization of monomial space is inadequate because it creates ridges that will fool the algorithm. Notice in Figure 2 that to get from an exponent of value k, to a value of $k+2$, one would need to consider an exponent of value $k+1$ which is of a very different ilk because its MSE will be so much higher than with k or $k+2$.

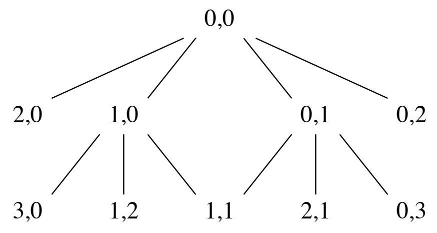

Figure 3. Two-Rule Monomial Space

To eliminate these ridges, we need a different organization of monomial space, called ‘Two-Rule Monomial Space’. We shall still place the 0-degree monomial at the root, but the children will be generated in a different order. There are now two generation rules:

1. For a monomial whose exponents are all no more than 1, change a 0 to a 1.
2. For any monomial, add 2 to any exponent.

Figure 3 shows the organization of monomial space that results. Notice that the ridges have been eliminated. Indeed, the four classes of monomial, oo, ee, oe, eo ('o' for odd and 'e' for even) are clearly visible below the root. Hereafter, adding only 2 to an exponent keeps the monomial in the same class.

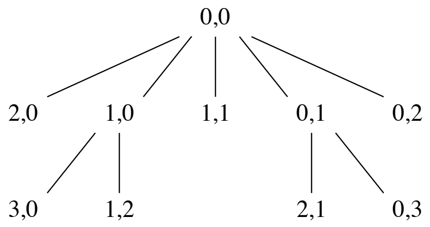

Figure 4. Three-Rule Monomial Space

The second serious problem is of a similar flavor, and will also be eliminated by reorganizing monomial space. There is a bias in variables of non-zero exponents. When an exponent is 0, the variable is effectively eliminated. To change an exponent from 0 to non-0 is to introduce a variable. As one moves in monomial space from a monomial of n variables to a monomial of $n+1$ variables, the number of variables changes, which can wreak havoc for negative values of the introduced variable because under some circumstances the function has been negated. This is much like parity. A ridge of a different sort is still present in monomial space. To fix this, we reorganize monomial space, giving 'Three-Rule Monomial Space', as defined by the three generation rules:

1. For the 0-degree monomial, change a 0 to a 1.
2. For a monomial whose exponents are all no more than 1, change any two exponents that are each 0 to 1.
3. For any monomial, add 2 to any exponent.

Figure 4 shows the new organization of monomial space that results. The parity ridges have been eliminated. The example would be more interesting for a higher number of variables. As it is, Rule 2 can only be applied once, as shown.

Let us consider how many children of the root there will be for d variables. We have d choices of 0 to change to a 1, and d choices of 0 to change to a 2, and $\frac{d(d-1)}{2}$ choices of pairs of 0s to change to 1s. At successive levels, application of Rule 2 will consume the 0s in the exponents. As soon as only Rule 3 applies, all the monomial classes have been generated. However, due to the greedy expansion, only those of apparent usefulness will be expanded.

Let us see how well the algorithm can do. Given our organization of monomial space, we can expect it to do well, assuming that appending monomials one at a time can work well.

3 Illustrations

Here is a trace of the GMG algorithm as it solves the third degree polynomial $f(\mathbf{x}) = 10x_0 + x_0^2 + 0.1x_0^3 + 2x_0x_1$ in two variables. The string of digits at the beginning of a line is the vector of exponents for that monomial. Commas have been omitted here because only single digit exponents appear in the traces shown. The 'best' tags were added by hand to help readability. The 2000 training points were each produced by generating random values within $[-10, 10]$ for each of the variables, and then evaluating the target function at that point.

```
Open List:
  00, mse =  13647.26064 <--- best
Append monomial:
  00, mse =  13647.26064

Open List:
  10, mse =   5356.14784 <--- best
  01, mse =  13646.40022
  11, mse =   9069.99210
  20, mse =  12913.98452
  02, mse =  13633.18004
Append monomial:
  10, mse =   5356.14784

Open List:
  01, mse =   5352.15427
  11, mse =   1058.87082 <--- best
  20, mse =   4493.51163
  02, mse =   5355.41690
  30, mse =   5172.61862
  12, mse =   5353.45464
Append monomial:
  11, mse =   1058.87082

Open List:
  01, mse =   1058.84724
  20, mse =    226.47090 <--- best
  02, mse =   1057.21293
  30, mse =    870.74560
  12, mse =   1053.48514
  31, mse =   1057.57780
  13, mse =   1058.04171
Append monomial:
  20, mse =    226.47090

Open List:
  01, mse =    226.46824
  02, mse =    226.15230
  30, mse =      0.00000 <--- best
  12, mse =    226.25875
  31, mse =    225.84713
  13, mse =    226.47083
  40, mse =    226.45717
  22, mse =    226.02626
Append monomial:
  30, mse =      0.00000

Approximation:
  30 *      0.10000 +
  20 *      1.00000 +
  11 *      2.00000 +
  10 *     10.00000 +
  00 *     -0.00000
```

We see that the algorithm finds the needed monomials quite directly. Note that the monomials appear one per line, with the coefficient following the vector of exponents. Only the 0-degree monomial was not needed, as evidenced by its coefficient of 0. Notice also how the MSE for the various monomials changes at each iteration. For example monomial $x_0^0 x_1^1$ (indicated by '01' in the trace) has MSE 13646 in iteration 1, but MSE 5352 in iteration 2. Similarly, monomial $x_0^3 x_1^0$ has MSE 5173 in iteration 3, even though it later has MSE 0. These dynamically changing values of MSE for nodes on the open guide the search at each step. GMG took 3 seconds for this problem on a 233MHZ Pentium II, including 37 linear regressions.

Let us now examine how GMG behaves on a target polynomial whose monomials are some distance from the root node. The target polynomial is in nine variables (seven relevant and two irrelevant) and includes a 10th degree term. It is $f(\mathbf{x}) = x_1^2 x_2^2 + x_0^3 + x_0 x_1 x_2 + x_0 x_1 + x_0 + x_1 + x_6^2 x_7^2 x_8^6 + x_1^2 x_5^4$. There were 2000 training points. The trace is lengthy, so only the final approximation is shown. From it, one can see all monomials that were absorbed into the polynomial. Those that were ultimately not needed have a coefficient of 0.

```
Approximation:
  010000000 *      1.00000 +
  111000000 *      1.00000 +
  110000000 *      1.00000 +
  300000000 *      1.00000 +
  100000000 *      1.00000 +
  022000000 *      1.00000 +
  020000000 *      0.00000 +
  022002000 *     -0.00000 +
  020004000 *      1.00000 +
  020002000 *      0.00000 +
  000002000 *      0.00000 +
  000000226 *      1.00000 +
  000000224 *      0.00000 +
  000000222 *     -0.00000 +
  000000202 *      0.00000 +
  000000002 *     -0.00000 +
  000000000 *      0.00001
```

We see that the eight needed monomials are all there, along with nine that were not needed. This is remarkably good efficiency considering that, as shown in Table 1, there are 92,378 monomials in the space. GMG took 261 seconds for this problem, including 1880 linear regressions.

Finally, here is an example of the algorithm finding a dense polynomial. By 'dense', we mean that all monomials up to the required degree are needed. This is in contrast to the two previous examples of 'sparse' polynomials. The target function is a five degree polynomial in two variables: $f(\mathbf{x}) = (x_0+0.1x_1-2)(2x_0-x_1-3)(0.3x_1-x_0+2)(2x_0+0.001x_1-3)(x_0-4x_1+1)$. There were 2000 training points. GMG took 43 seconds for this problem, including 262 linear regressions.

```
Approximate Function:
  05 *      0.00012 +
  04 *     -0.36127 +
  03 *      3.78951 +
  02 *     61.42720 +
  14 *      0.24053 +
  13 *     -4.10823 +
  12 *   -106.21020 +
  23 *      1.05426 +
  50 *     -4.00000 +
  40 *     24.00000 +
  22 *     60.71200 +
  41 *     18.79800 +
  31 *   -124.19100 +
  20 *     11.00000 +
  32 *    -11.47060 +
  30 *    -45.00000 +
  11 *   -325.00800 +
  21 *    303.59100 +
  01 *    128.41200 +
  10 *     48.00000 +
  00 *    -36.00000
```

4 Related Work

Sutton and Matheus (1991) proposed a polynomial function approximation algorithm that appends monomials one at a time. Their algorithm initializes the open list with all the 1-degree monomials. Thereafter, a new successor monomial can be produced by forming the product of two monomials already on the open list. Monomials are not removed from the open list, allowing every monomial to remain a candidate for forming a new monomial. To guide selection of a pair of monomials to use in constructing a new one, they compute a measure, called the 'potential', for each existing monomial. The two monomials with the highest potentials are selected and combined. The potential of a monomial is the coefficient of the square of that monomial when finding a least mean-squared-error fit for all the squares of the existing monomials. They provide a further improvement, called a 'joint potential', which is the potential of a cross term (product of two monomials).

We implemented their algorithm. Rich Sutton kindly provided his Lisp code, but for a variety of reasons, we wrote our own C code. Our implementation reproduces exactly all the examples reported by Sutton and Matheus. We have found their algorithm to work quite well for monomials that have variables with low exponents, but that it can miss finding monomials with variables of high exponents. We suspect that this is due to ridges in their monomial space. For example, their algorithm does not find an approximation for the first example problem above. After generating the monomial x_0^2 and $2x_0x_1$ (x_0 is already there initially), the program could not find x_0^3. After the algorithm finds x_0^2, the potential of x_0 becomes negative, and stays negative indefinitely, which means it is not selected for combination.

Numerical Recipes (Press, Flannery, Teukolsky & Vetterling, 1988) contains a variety of methods for polynomial interpolation and extrapolation in one variable. For multi-dimensional polynomial construction, they suggest two local methods. One is to build a multi-dimensional grid of approximating functions, and the other is to slice the multi-dimensional space recursively, grounding out in a multitude of one-dimensional approximation problems.

MATLAB (Borse, 1997) offers a polynomial fitting function, but the user must specify the degree The algorithm enumerates the monomials of a dense polynomial of that degree, and computes the coefficients that minimize the mean-squared error. This approach is impractical because of the sea of monomials that will be generated for functions of many variables or of high degree.

5 Summary

We have presented a practical algorithm GMG for finding polynomial functions of high dimension and high degree. The strength of the algorithm comes from the organization of monomial space in a manner that removes ridges in the error function. The algorithm is able to solve approximation problems that were previously not solvable automatically. The algorithm is oriented toward polynomial approximation, but the search algorithm and the notion of space reorganization may be extendible to other model classes.

Acknowledgments

This work was supported by National Science Foundation Grant IRI-9711239. We are indebted to John Buonaccorsi of the UMass Mathematics and Statistics Department, who provided many useful comments about polynomial regression in the Statistics community. Rich Sutton kindly supplied his program code. We thank Gang Ding and David Stracuzzi for helpful comments.

References

Borse, G. J. (1997). *Numerical methods with MATLAB*. PWS Publishing Company.

Press, W. H., Flannery, B. P., Teukolsky, S. A., & Vetterling, W. T. (1988). *Numerical recipies in C: The art of scientific computing*. New York: Cambridge University Press.

Reingold, E. M., Nievergelt, J., & Deo, N. (1977). *Combinatorial algorithms: Theory and practice*. Prentice Hall.

Sutton, R. S., & Matheus, C. J. (1991). Learning polynomial functions by feature construction. *Machine Learning: Proceedings of the Eighth International Workshop* (pp. 208-212). Evanston, IL: Morgan Kaufmann.