Bernstein Polynomials as Fuzzy Indicators

Jan de Leeuw

First created on May 04, 2022. Last update on May 04, 2022

Abstract

Optimal scaling methods transform variables by using a basis for the subspace or a frame for the cone of admissible transformations. In this note we argue that for polynomials and monotone polynomials the Bernstein basis should be used.

1 Introduction

The more recent versions of the Gifi system for descriptive multivariate analysis have the option to replace the crisp binary indicator matrices that form a basis for the space of quantifications or transformations of a variable by fuzzy indicators (Rijckevorsel and De Leeuw (1988), De Leeuw and Mair (2009)). A fuzzy indicator is a non-negative matrix with all its row sums equal to one. A crisp indicator is the special case in which the entries are all zero or one.

Currently in the Gifi programs the fuzzy indicators are computed by using a B-spline basis for the piecewise polynomials on a given knot sequence. Fitting polynomial transformations is still done by using a basis of monomials, which generally does not produce an indicator. Both splines and polynomials can be restricted to be non-negative, monotone, and convex or concave.

In this note we propose the Bernstein polynomials as an alternative basis for the polynomials of a fixed degree. They have the major advantage that they define a fuzzy indicator, and, like the B-splines, they are shape-preserving, which means they reproduce monotonicity and convexity/concavity.

2 Bernstein Polynomials

A *Bernstein polynomial* of degree n at $0 \leq x \leq 1$ is of the form

$$\beta_n(x) = \sum_{k=0}^{n} \alpha_k b_{n,k}(x) \tag{1}$$

where the $(n+1)$ polynomials

$$b_{n,k}(x) := \binom{n}{k} x^k (1-x)^{n-k} \tag{2}$$

are the Bernstein basis of decree n at x (Lorenz (1953), Bustamante (2017)).

Statisticians and mathematicians familiar with the binomial distribution will see at a glance from (2) that $0 \leq b_{n,k}(x) \leq 1$. In addition $b_{n,k}(x) = 0$ if and only if $x = 0$ and $b_{n,k}(x) = 1$ if and only if $x = 1$. Moreover

$$\sum_{k=0}^{n} b_{n,k}(x) = 1. \tag{3}$$

Thus a Bernstein basis defines a fuzzy indicator.

Here is a one-line R function to compute a Bernstein basis.

```
bernsteinR <- function (n, x, k = 0:n) {
  return(outer (x, k, function(a, b) dbinom(b, n, a)))
}
```

For the eleven equally spaced points between zero and one (inclusive), and for degree 3, bernsteinR() gives

```
##        [,1]  [,2]  [,3]  [,4]
##  [1,] 1.000 0.000 0.000 0.000
##  [2,] 0.729 0.243 0.027 0.001
##  [3,] 0.512 0.384 0.096 0.008
##  [4,] 0.343 0.441 0.189 0.027
##  [5,] 0.216 0.432 0.288 0.064
##  [6,] 0.125 0.375 0.375 0.125
##  [7,] 0.064 0.288 0.432 0.216
##  [8,] 0.027 0.189 0.441 0.343
##  [9,] 0.008 0.096 0.384 0.512
## [10,] 0.001 0.027 0.243 0.729
## [11,] 0.000 0.000 0.000 1.000
```

The corresponding Vandermonde matrix with monomials is

```
##       [,1] [,2] [,3]  [,4]   [,5]
##  [1,]    1  0.0 0.00 0.000 0.0000
##  [2,]    1  0.1 0.01 0.001 0.0001
##  [3,]    1  0.2 0.04 0.008 0.0016
##  [4,]    1  0.3 0.09 0.027 0.0081
##  [5,]    1  0.4 0.16 0.064 0.0256
##  [6,]    1  0.5 0.25 0.125 0.0625
```

```
##  [7,]    1  0.6 0.36 0.216 0.1296
##  [8,]    1  0.7 0.49 0.343 0.2401
##  [9,]    1  0.8 0.64 0.512 0.4096
## [10,]    1  0.9 0.81 0.729 0.6561
## [11,]    1  1.0 1.00 1.000 1.0000
```

Both matrices span the same space, the four dimensional space of third degree polynomials. One important difference between the two is that the Bernstein basis is better conditioned than the Vandermonde basis. The ratio of the largest to the smallest singular value of the two matrices above is 4.7723371457 for Bernstein and 59.3359273583 for Vandermonde.

Another, and perhaps even more important, difference in the Gifi (and any other optimal scaling) context is that the polynomial β_n from (1) is monotone increasing if $\alpha_0 \leq \alpha_1 \leq \cdots \leq \alpha_n$ and convex if $\alpha_{k+1} \leq \frac{1}{2}(\alpha_k + \alpha_{k+2})$ for all $k = 0, \cdots, n-2$ (or, equivalently, $\alpha_k - \alpha_{k+1} \leq \alpha_{k+1} - \alpha_{k+2}$. For monotone decreasing and concave just reverse the inequalities.

One possible problem with computing the Bernstein basis is that x must be in the closed unit interval. We can deal with that in two ways. First, we can linearly rescale the variable to be transformed so that it is between zero and one. A third degree polynomial of a linear transformed variable is still a third degree polynomial. Linear rescaling also preserves monotonicity and convexity. Secondly, and actually equivalently, we could redefine the Bernstein polynomials so that they can be computed for any closed interval $[a, b]$. Note that that the Bernstein basis is a basis for *all* polynomials of a fixed degree, not just for polynomials on a finite interval $[0, 1]$ (or $[a, b]$).

3 C Program

The computation in R of the Bernstein basis, although nice and compact, may not be very efficient. First, because it is in R, although the function we gave does not contain loops and mainly consists of calls to compiled C routines. Secondly, because evaluating many binomial cofficients can be done more efficiently by the recursive algorithm

$$b_{k,n}(x) = (1-x)b_{k,n-1}(x) + xb_{k-1,n-1}(x) \tag{4}$$

So we wrote the function bernsteinC() in C, in addition to R glue bernsteinRC() that calls the C routine from R. Code is in the appendix.

Timing for a case with 10001 rows and degree equal to 100 suggests that in this (unrealistically) large example the C code can be up to two times as fast.

```
x <- seq(0, 1, length = 10001L)
microbenchmark(bernsteinR(100L, x), bernsteinRC(100L, x))
```

```
## Unit: milliseconds
##                  expr       min         lq         mean      median         uq
```

```
##   bernsteinR(100L, x) 89.481986 96.3420930 104.29504663 101.7340405 109.074921
##  bernsteinRC(100L, x) 42.488728 47.9699705  54.75641064  52.0734435  56.832769
##         max neval
##  149.341235   100
##  115.959002   100
```

For 1000 rows and a polynomial of degree 10 we find a somewhat larger speedup factor.

```
x <- seq(0, 1, length = 1001L)
microbenchmark(bernsteinR(10L, x), bernsteinRC(10L, x))
```

```
## Unit: microseconds
##                 expr     min        lq       mean    median       uq      max
##   bernsteinR(10L, x) 861.011 1107.1415 1325.52097 1281.6085 1512.491 2193.515
##  bernsteinRC(10L, x) 200.180  303.7880  455.67482  426.7385  586.075  993.073
##  neval
##    100
##    100
```

For a tiny example with 11 rows and a polynomial of degree 4 the code in R actually wins, because for this case the overhead of the .C() interface dominates the computing time.

```
x <- seq(0, 1, length = 11L)
microbenchmark(bernsteinR(4L, x), bernsteinRC(4L, x), times = 1000)
```

```
## Unit: microseconds
##                expr    min     lq      mean median      uq     max neval
##   bernsteinR(4L, x) 10.498 19.891 32.241239 27.642 38.0650 146.987  1000
##  bernsteinRC(4L, x) 32.299 50.527 68.935569 61.492 77.9025 242.467  1000
```

A Glue in R

```
dyn.load("bernstein.so")

bernsteinRC <- function (d, x) {
  n <- length(x)
  m <- n * (d + 1L)
  h <- .C(
    "bernsteinC",
    d = as.integer(d),
```

```
    n = as.integer(length(x)),
    x = as.double(x),
    y = as.double(rep(0, m))
  )
  return(matrix(h$y, n, d + 1))
}
```

B Code in C

```
void bernsteinC(int *pd, int *pn, double *px, double *py);

#define VINDEX(i) ((i)-1)
#define MINDEX(i, j, n) ((i)-1) + ((j)-1) * (n)

/*****************************************************************************/

void bernsteinC(int *pd, int *pn, double *px, double *py) {
    int n = *pn, d = *pd;
    double x = 0;
    for (int k = 1; k <= n; k++) {
        x = px[VINDEX(k)];
        if (d == 0) {
            py[MINDEX(k, 1, n)] = 1.0;
        } else if (0 < d) {
            py[MINDEX(k, 1, n)] = (1 - x);
            py[MINDEX(k, 2, n)] = x;

            for (int i = 3; i <= (d + 1); i++) {
                py[MINDEX(k, i, n)] = x * py[MINDEX(k, i - 1, n)];
                for (int j = i - 1; 2 <= j; j--) {
                    py[MINDEX(k, j, n)] = ((1 - x) * py[MINDEX(k, j, n)] +
                                           x * py[MINDEX(k, j - 1, n)]);
                }
                py[MINDEX(k, 1, n)] = (1 - x) * py[MINDEX(k, 1, n)];
            }
        }
    }
    return;
}
```

References

Bustamante, J. 2017. *Bernstein Operators and Their Properties*. Birkhauser.

De Leeuw, J., and P. Mair. 2009. "Homogeneity Analysis in R: the Package homals." *Journal of Statistical Software* 31 (4): 1–21.

Lorenz, G. G. 1953. *Bernstein Polynomials*. University of Toronto Press.

Rijckevorsel, J. L. A. Van, and J. De Leeuw, eds. 1988. *Component and Correspondence Analysis*. Wiley.