Minimum of the Interpolating Polynomial with Applications in Cyclic Coordinate Descent

Jan de Leeuw

First created on April 24, 2020. Last update on October 29, 2021

Abstract

Given two vectors x and y of the same length n, the univariate interpolating polynomial p of degree $n-1$ has $p(x_i) = y_i$. We give code in C and R to compute the minimum of this interpolating polynomial, which is useful in cyclic coordinate descent algorithms. Various applications in data analysis are discussed.

Contents

Note: This is a working paper which will be expanded/updated frequently. All suggestions for improvement are welcome. The directory deleeuwpdx.net/pubfolders/poly has a pdf version, the bib file, the complete Rmd file with the code chunks, and the R and C source code.

1 Introduction

Many of the least squares loss functions used in multivariate analysis are multivariate polynomials of the parameters. This is true for principal component analysis, independent component analysis (Hyvärinen, Karhumen, and Oja (2001)), three-way analysis (Kroonenberg and De Leeuw (1980)), cumulant component analysis (Lim and Morton (2008)), squared-distance multidimensional scaling (Takane, Young, and De Leeuw (1977)), moment component analysis (Jondeau, Jurczenko, and Rockinger (2018)), and factor analysis (Harman and Jones (1966)). It implies, of course, that if we change a single parameter and keep all others fixed at their current value, then loss is a univariate polynomial. Since least squares loss functions are non-negative this univariate polynomial attains its minimum. This makes it possible to apply cyclic coordinate descent (CCD), changing one parameter at a time and cycling through all parameters in a systematic way.

The resulting algorithm does not require the calculation of multivariate derivatives of the orginal loss function and is generally convergent to at least a local minimum. Convergence is bound to be slow, but usually not so slow that the algorithm becomes useless.

This paper provides some of the necessary tools to implement a CCD algorithm on these multivariate polynomials. We evaluate loss at an appropriate number of values different from the current one, and compute the interpolating polynomial. Thus we do not need the explicit mathematical form of the univariate polynomial, we compute the coefficients by solving the corresponding Vandermonde system. We then differentiate the interpolating polynomial and compute the roots of the derivative. And finally we select the real root which gives the smallest loss.

2 Computation

2.1 Conventions

Our C code follows the conventions used in a bunch of other publications. The routines can be called from the .C() interface in R, which means that they return void and always pass by reference. They do not use matrix and vector indexing via [ ], but instead use pointer arithmetic and an inline function VINDEX() which computes the appropriate place in memory.

2.2 Interpolating Polynomial

The interpolating polynomial is computed by the dvand() function of Björck and Pereyra (1970), previously implemented in C by Burkardt (2019). We obviously needed some trivial modifications because of our general coding conventions.

2.3 Roots of the Derivative

To compute roots of polynomials we use the GSL library (Galassi et al. (2019)), in particular the section on general polynomial equations. The function polysolve() is a C wrapper for gsl_poly_complex_solve(). We also use GSL to evaluate polynomials, with polyval() wrapping gsl_poly_eval(). We could get rid of the wrappers and eliminate some function calls, but we have written polysolve() and polyval() using the .C() conventions.

2.4 Compilation

The C code is compiled to a shared library with the command

```
R CMD SHLIB poly.c -lgsl
```

This obviously assumes that GSL is installed on your system.

2.5 Tying it together

The C function intermin() calls dvand(), polysolve(), and polyval() in succession to find the minimum of the interpolating polynomial. The R function inC() is a wrapper for intermin().

2.6 Example

```
a<-1:5
b<-(a-1)*(a-2)*(a-3)*(a-4)
```

Obviously these are 5 points on a fourth degree polynomial with roots 1,2,3,4.

```
inC(a, b)
```

```
## [1] 1.381966011
```

We can check our result with the polynom package in R (Venables, Hornik, and Maechler (2019)), using the wrapper inR() which calls the C function dvand() to computing the interpolating polynomal, after which the polynom package takes over.

```
inR(a,b)
```

```
## [1] 3.618033989
```

Initially it may be somewhat disconcerting that the two wrappers give a different result. But the following sequence of R calls shows what is going on.

```
p<-polynomial(c(-1,1))*polynomial(c(-2,1))*polynomial(c(-3,1))*polynomial(c(-4,1))
deriv(p)
```

```
## -50 + 70*x - 30*x^2 + 4*x^3
```

```
solve(deriv(p))
```

```
## [1] 1.381966011 2.500000000 3.618033989
```

```
predict(p,solve(deriv(p)))
```

```
## [1] -1.0000  0.5625 -1.0000
```

Thus the polynomial, which is of course the interpolating polynomial, has two equal minima, one at 1.3819660113 and one at 3.6180339887. No problemo.

3 Applications

3.1 PCA/LSFA

Consider the loss function

$$\sigma(X) = \sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}(r_{ij} - \sum_{s=1}^{p} x_{is}x_{js})^2 \tag{1}$$

where $W = \{w_{ij}\}$ and $R = \{r_{ij}\}$ are given matrices, with $w_{ij} \geq 0$ for all i,j. This is weighted low-rank matrix approximation, which can be used for principal component analysis, matrix completion, and for factor analysis (if $w_{ii} = 0$ for all i). Because loss is a fourth degree polynomial we need five interpolation points.

To illustrate how CCD works on a real example, we use the Harman.8 data from the psych package (Revelle (2018)). This is a correlation matrix between 8 variables, and we choose W as the hollow matrix defining least squares factor analysis.

The program ccd() is written in R. The function loss() computes the loss, and as a bonus we also output the estimated communalities diag(XX'). Of course we could speed up the computations considerably by writing ccd() in C as well.

The function ccd() can be used as a template, because other applications of the same general algorithm often need only minor modifications, mainly by defining the appropriate loss() and by choosing suitable interpolation points (about which later).

```
data(Harman.8, package = "psych")
r <- Harman.8
w <- 1 - diag(8)
e <- eigen(r)
x <- e$vectors[,1:2]%*%diag(sqrt(e$values[1:2]))

loss <- function (w, r, x) {
  s <- sum (w * (r - tcrossprod(x)) ^ 2)
  return(s)
}

ccd <- function (w, r, x, itmax = 100, eps = 1e-8, verbose = TRUE) {
  n <- nrow (x)
  p <- ncol (x)
  fold <- loss (w, r, x)
  pert <- c(-.01,-.02,0,.01,.02)
  fval <- pert
  itel <- 1
  repeat {
    for (i in 1:n) {
```

```
      for (s in 1:p) {
        for (k in 1:length(pert)) {
          fval[k] <- loss(w, r, x + pert[k])
        }
        x[i, s] <- x[i, s] + inC(pert, fval)
      }
    }
    fnew <- loss(w, r, x)
    if (verbose) {
      cat("itel ", formatC(itel, width = 3, format = "d"),
          "fold ", formatC(fold, width = 10, digits = 8, format = "f"),
          "fnew ", formatC(fnew, width = 10, digits = 8, format = "f"),
          "\n")
    }
    if (((fold - fnew) < eps) || (itel == itmax)) {
      break
    }
    fold <- fnew
    itel <- itel + 1
  }
  return (list (f = fnew, x = x, d = diag(tcrossprod(x))))
}

ccd (w, r, x)
```

```
## itel    1 fold  0.15486142 fnew  0.13806151
## itel    2 fold  0.13806151 fnew  0.13585058
## itel    3 fold  0.13585058 fnew  0.13550850
## itel    4 fold  0.13550850 fnew  0.13544053
## itel    5 fold  0.13544053 fnew  0.13542303
## itel    6 fold  0.13542303 fnew  0.13541769
## itel    7 fold  0.13541769 fnew  0.13541592
## itel    8 fold  0.13541592 fnew  0.13541532
## itel    9 fold  0.13541532 fnew  0.13541511
## itel   10 fold  0.13541511 fnew  0.13541504
## itel   11 fold  0.13541504 fnew  0.13541502
## itel   12 fold  0.13541502 fnew  0.13541501

## $f
## [1] 0.1354150099
##
## $x
##                [,1]          [,2]
## [1,] -0.8425728477 -0.3577634559
```

```
## [2,] -0.8271298436 -0.4286591618
## [3,] -0.8007446751 -0.4479848680
## [4,] -0.8289621134 -0.3862174054
## [5,] -0.7489417097  0.5330077106
## [6,] -0.6661209743  0.5407717530
## [7,] -0.6099307830  0.5868932475
## [8,] -0.6640176777  0.4244900570
##
## $d
## [1] 0.8379236941 0.8678924552 0.8418824767 0.8363420697 0.8450109041
## [6] 0.7361512412 0.7164592440 0.6211112847
```

3.2 MDS

For squared distance MDS the loss is

$$\sigma(X) = \sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}(\delta_{ij} - \sum_{s=1}^{p}(x_{is} - x_{js})^2)^2. \tag{2}$$

Again this is a fourth degree polynomial, and basically the same algorithm can be used as for PCA/LSFA.

3.3 Multiway

The general least squares multiway loss function is

$$\sigma(X^{\{1\}}, \cdots, X^{\{m\}}) = \sum_{i_1=1}^{n_1} \cdots \sum_{i_m=1}^{n_m} w_{i_1\cdots i_m}(z_{i_1\cdots i_m} - \sum_{s_1=1}^{p_1} \cdots \sum_{s_r=1}^{p_r} k_{s_1\cdots s_r} x_{i_1 s_1}^{\{1\}} \cdots x_{i_m s_r}^{\{m\}})^2. \tag{3}$$

This is quadratic in each of the parameters. Obviously in this case CCD programs to minimize loss will need a lot of index manipulation (De Leeuw (2017)). Note that a minor variation requires the core K to be diagonal, and absorbs it into the $X^{\{j\}}$, thus generalizing INDSCAL (Carroll and Chang (1970)).

Matters become more interesting in the supersymmetric case, in which all $X^{\{j\}}$ are required to be equal, making loss a polynomial of degree $2m$. This is needed for cumulant component analysis and will be implemented in a subsequent report.

4 Discussion

There are two additional points worth noting. First CCD methods for minimizing multivariate polynomials can easily be modified to take simple constraints on the parameters into account. In confirmatory factor analysis (Harrington (2009)), for example, some elements of

X are kept at fixed values (often at zero), which means that we simply skip them in CCD cycles. In non-negative matrix factorization (Naik (2016)) the parameters are required to be non-negative, so our univariate polynomials must have non-negative minimizers (which means that we also have to consider an additional boundary solution where the modified parameter becomes zero). Again, this does not really complicate the CCD algorithm.

The most interesting point, and one we have not really touched on, is the choice of the interpolation points. The number is fixed by the degree of the polynomial, but the choice should probably be determined by the scale of the parameters. Mathematically the choice does not make a difference, but computationally we do not want our Vandermonde systems to be too ill-conditioned, because that will lead to a possibly serious loss of precision.

5 Appendix: Code

5.1 R Code

5.1.1 poly.R

```
dyn.load("poly.so")
library(polynom)

inC <- function (a, b) {
  degree = length(a) - 1
  h <-
    .C(
      "intermin",
      degree = as.integer(degree),
      a = as.double (a),
      b = as.double (b),
      res = as.double (0)
    )
  return (h$res)
}

inR <- function (a, b) {
  degree <- length(a) - 1
  h <-
    .C(
      "dvand",
      order = as.integer(degree + 1),
      a = as.double(a),
      b = as.double(b),
      x = as.double(b)
```

```
    )
  p <- polynomial(h$x)
  q <- deriv(p)
  s <- solve(q)
  r <- s[!is.complex(s)]
  v <- predict(p, r)
  return (r[which.min(v)])
}
```

5.2 C code

5.2.1 poly.h

```
#ifndef POLY_H
#define POLY_H

#include <gsl/gsl_poly.h>
#include <math.h>

inline int VINDEX(const int i) { return i - 1; }

#endif /* POLY_H */
```

5.2.2 poly.c

```
#include "poly.h"

void polysolve(const double *a, const int *pdegree, double *z) {
  int degree = *pdegree;
  gsl_poly_complex_workspace *w = gsl_poly_complex_workspace_alloc(degree + 1);
  gsl_poly_complex_solve(a, degree + 1, w, z);
  gsl_poly_complex_workspace_free(w);
  return;
}

void polyval(const int *pdegree, const double *a, const double *x, double *s) {
  int degree = *pdegree;
  *s = gsl_poly_eval(a, degree + 1, *x);
  return;
}
```

```
void dvand(const int *pn, const double *a, const double *b, double *x) {
  int n = *pn;
  for (int i = 1; i <= n; i++) {
    *(x + VINDEX(i)) = *(b + VINDEX(i));
  }
  for (int k = 1; k < n; k++) {
    for (int j = n; k < j; j--) {
      *(x + VINDEX(j)) = (*(x + VINDEX(j)) - *(x + VINDEX(j - 1))) /
                         (*(a + VINDEX(j)) - *(a + VINDEX(j - k)));
    }
  }
  for (int k = n - 1; 1 <= k; k--) {
    for (int j = k; j < n; j++) {
      *(x + VINDEX(j)) =
          *(x + VINDEX(j)) - *(a + VINDEX(k)) * *(x + VINDEX(j + 1));
    }
  }
  return;
}

void intermin(const int *pdegree, const double *a, const double *b,
              double *res) {
  int degree = *pdegree, length = degree + 1, nroots = degree - 1;
  double *x = (double *)calloc((size_t)length, sizeof(double));
  double *y = (double *)calloc((size_t)degree, sizeof(double));
  double *z = (double *)calloc((size_t)(2 * nroots), sizeof(double));
  (void)dvand(&length, a, b, x);
  for (int i = 1; i <= degree; i++) {
    *(y + VINDEX(i)) = *(x + VINDEX(i + 1)) * ((double)i);
  }
  (void)polysolve(y, &nroots, z);
  double mval = INFINITY, xval, zval, val;
  for (int i = 1; i <= nroots; i++) {
    if (fabs(z[VINDEX(2 * i)]) < 1e-15) {
      zval = z[VINDEX((2 * i) - 1)];
      (void)polyval(&length, x, &zval, &val);
      if (val < mval) {
        mval = val;
        xval = zval;
      }
    }
  }
  *res = xval;
  (void)free(x);
```

```
  (void)free(y);
  (void)free(z);
  return;
}
```

References

Björck, Å., and V. Pereyra. 1970. "Solution of Vandermonde Systems of Equations." *Mathematics of Computation* 24: 893–903.

Burkardt, J. 2019. "VANDERMONDE. Accurate Solution of Vandermonde Systems." 2019. https://people.sc.fsu.edu/~jburkardt/c_src/vandermonde/vandermonde.html.

Carroll, J. D., and J. J. Chang. 1970. "Analysis of Individual Differences in Multidimensional scaling via an N-way generalization of "Eckart-Young" Decomposition." *Psychometrika* 35: 283–319.

De Leeuw, J. 2017. "Multidimensional Array Indexing and Storage." 2017.

Galassi, M., J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, F. Rossi, and R. Ulerich. 2019. "GNU Scienfific Library, Reference Manual." In, Edition 2.6. https://www.gnu.org/software/gsl/doc/html/.

Harman, H. H., and W. H. Jones. 1966. "Factor Analysis by Minimizing Residuals ." *Psychometrika* 31: 351–68.

Harrington, D. 2009. *Confirmatory Factor Analysis*. Oxford University Press.

Hyvärinen, A., J. Karhumen, and E. Oja. 2001. *Independent Component Analysis*. Wiley.

Jondeau, E., E. Jurczenko, and M. Rockinger. 2018. "Moment Component Analysis: An Illustration With International Stock Markets." *Journal of Business & Economic Statistics* 36: 576–98.

Kroonenberg, P. M., and J. De Leeuw. 1980. "Principal Component Analysis of Three-Mode Data by Means of Alternating Least Squares Algorithms." *Psychometrika* 45: 69–97.

Lim, L.-H., and J. Morton. 2008. "Cumulant Component Analysis: A Simultaneous Generalization of PCA and ICA." In *Computational Algebraic Statistics. Theories and Applications*. Kyoto.

Naik, G. R. 2016. *Non-Negative Matrix Factorization Techniques*. Springer Verlag.

Revelle, W. 2018. *psych: Procedures for Psychological, Psychometric, and Personality Research*. Evanston, Illinois: Northwestern University. %7Bhttps://CRAN.R-project.org/package=psych%7D.

Takane, Y., F. W. Young, and J. De Leeuw. 1977. "Nonmetric Individual Differences in Multidimensional Scaling: An Alternating Least Squares Method with Optimal Scaling Features." *Psychometrika* 42: 7–67. http://www.stat.ucla.edu/~deleeuw/janspubs/1977/articles/takane_young_deleeuw_A_77.pdf.

Venables, W. N., K. Hornik, and M. Maechler. 2019. "polynom: A Collection of Functions to Implement a Class for Univariate Polynomial Manipulations." 2019. https://CRAN.R-project.org/package=polynom.