Linear Regression by Folding

Brian Beckman
12 Feb 2018

Introduction

Linear systems appear everywhere, and, where they don’t appear naturally, linear approximations abound because non-linear systems are often intractable. Examples comprise machine learning, control systems, dynamics, robotics, and many more.

Linear regression is the standard technique for estimating the parameters or coefficients of a linear system. Usually, authors sweep linear regression under the rug, presumably because readers know all about effective ways to perform it. However, I have found this presumption not to be true. Time and again, I see code that employs the normal equations directly (that's inefficient), computes matrix inverses (that's risky), or abuses neural nets (a sledgehammer) to open a walnut (linear regression).

This paper shows an efficient, reliable method for very general linear regression. The method is mathematically equivalent to a Kalman fold for static systems. It is amongst the best known methods for linear regression.

Motivating Example

Find best-fit, unknowns m, b, where $z \approx m\,x + b$, given known data $(z_1, z_2, \ldots, z_k)$ and $(x_1, x_2, \ldots, x_k)$. Write this *system* as a matrix equation and remember the symbols Z (observations, known), A (partials, known), and Ξ (*model, state, unknown parameters to be estimated*). Rows of Z and A come in matched pairs.

$$Z_{k\times 1} = \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_k \end{pmatrix} = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_k & 1 \end{pmatrix} \cdot \begin{pmatrix} m \\ b \end{pmatrix} + \text{noise} = A_{k\times 2} \cdot \Xi_{2\times 1} + \text{samples of NormalDistribution}[0,\ \sigma_Z] \qquad (1)$$

This extends to any linear system with any number of parameters and with tensors in the matrix slots.

Ground Truth

Make some data by sampling a line specified by ground truth, then adding some noise. Run the faked data through our estimation procedures and see how close the estimates come to the ground truth. This procedure tests the method. In the real world, we don't know ground truth, so it's paramount that we trust the method before deploying it. Build up trust by testing.

```
ClearAll[groundTruth, m, b];
groundTruth = {m, b} = {0.5, -1. / 3.};
```

Partials

The partials *A* are a (column) vector of co-vectors (row vectors). Each co-vector is the gradient of the observation model, namely of *A*·Ξ, with respect to Ξ, evaluated at a specific Ξ. Gradients of vector functions are co-vectors, that is, linear transformations of vectors. This fact becomes interesting when considering the dual problem, below.

```
ClearAll[nData, min, max];
nData = 119; min = -1.; max = 3.;
ClearAll[partials];
partials = Array[{#, 1.0} &, nData, {min, max}];
Short[partials, 3]
```

```
{{-1., 1.}, {-0.966102, 1.}, ≪115≫, {2.9661, 1.}, {3., 1.}}
```

Fake Data

```
ClearAll[fake];
fake[n_, σ_, A_, {m_, b_}] :=
  Table[
   RandomVariate[NormalDistribution[0, σ]] + A⟦i⟧.{m, b},
   {i, n}];
```

```
ClearAll[data, noiseSigma];
noiseSigma = 0.65;
data = fake[nData, noiseSigma, partials, groundTruth];
Short[data, 3]
```

```
{-1.40787, -1.4709, -1.06779, ≪114≫, 1.92515, 0.672254}
```

Model

Try the Wolfram built-in. The estimated *m* and *b* are reasonably close to the ground truth.

```
ClearAll[model];
model = LinearModelFit[{partials⟦All, 1⟧, data}ᵀ, ξ, ξ];
Normal[model]
```

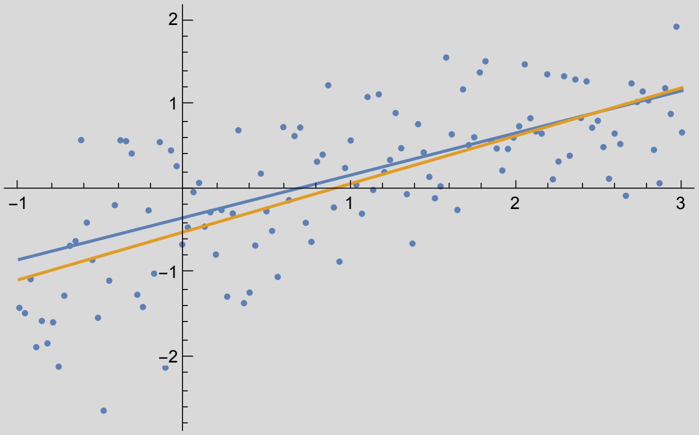
```
-0.503288 + 0.568383 ξ
```

Un-comment the following line to see everything Wolfram has to say about this model (it’s a lot of data).

```
(*Association[(#→model[#])&/@model["Properties"]]*)
```

The plot shows that Wolfram does an acceptable job of estimating the parameters *m* and *b* that define the line:

```
Show[ListPlot[{partials⟦All, 1⟧, data}ᵀ],
 Plot[{m ξ + b, model[ξ]}, {ξ, min, max}]]
```



In the following, we show several methods that are all mathematically equivalent to each other, but have vastly different operational characteristics regarding memory usage and numerical risk.

Normal Equations

Equation 1 can be solved for a value of Ξ that minimizes the square error $J(\Xi) \overset{\text{def}}{=} (Z - A\cdot\Xi)^{\intercal}\cdot(Z - A\cdot\Xi)$. Because the noise $\mathcal{N}(0, \sigma)$ has zero mean (or zero *bias*), The solution turns out to be exactly what one would get from naive algebra. $A^{\intercal}\cdot A$ is square. When it is invertible,

$$(A^{\intercal}\cdot A)^{-1}\cdot A^{\intercal}\cdot Z \;=\; \Xi \tag{2}$$

That gives the same answer as Wolfram’s built-in:

```
Inverse[partialsᵀ.partials].partialsᵀ.data
```

```
{0.568383, -0.503288}
```

The matrix $(A^{\mathsf{T}} \cdot A)^{-1} \cdot A^{\mathsf{T}}$ is the *Moore-Penrose left pseudoinverse*. Wolfram has a built-in for it:

```
PseudoInverse[partials].data
```

```
{0.568383, -0.503288}
```

$(A^{\mathsf{T}} \cdot A)^{-1} \cdot A^{\mathsf{T}} \cdot Z$ is a very nasty computation: in memory usage, in time, and in numerical risk. Eliminate these defects with a recurrence. This recurrence is mathematically identical to a Kalman filter in parameter-estimation mode, though we do not prove that here.

Recurrence

Fold the following recurrence over Z and A:

$$\begin{aligned} \psi &\leftarrow (\Lambda + a^{\mathsf{T}} \cdot a)^{-1} \cdot (a^{\mathsf{T}} \cdot \zeta + \Lambda \cdot \psi) \\ \Lambda &\leftarrow (\Lambda + a^{\mathsf{T}} \cdot a) \end{aligned} \tag{3}$$

where ψ is the current estimate of Ξ, a and ζ are matched rows of A and Z, and Λ accumulates $A^{\mathsf{T}} \cdot A$.

Derive the recurrence as follows: Treat the estimate-so-far, ψ, as just one more observation with information matrix (proportional to inverse covariance) $\Lambda = A_{\text{so-far}}{}^{\mathsf{T}} \cdot A_{\text{so-far}}$. The scalar *performance* or *squared error* of the estimate, so far, is $J(x) = (z - A_{\text{so-far}} \cdot x)^{\mathsf{T}} \cdot (z - A_{\text{so-far}} \cdot x) = (x - \psi)^{\mathsf{T}} \cdot \Lambda \cdot (x - \psi)$, where x is the unknown true parameter vector and $\Lambda = A^{\mathsf{T}} \cdot A$. Adding a new observation, ζ and its corresponding partial a, increases the error by $(\zeta - a \cdot x)^{\mathsf{T}} \cdot (\zeta - a \cdot x)$. Minimizing the new total error with respect to x yields the recurrence.

The initial value of ψ does not matter much, but the initial value of Λ cannot be singular. For practical purposes, any Λ_0 with terms much smaller than typical terms in $A_{\text{so-far}}{}^{\mathsf{T}} \cdot A_{\text{so-far}}$ will do. The example below starts with $\psi_0 = (\,0 \quad 0\,)^{\mathsf{T}}$ and $\Lambda_0 = \left(\left(\,10^{-6} \quad 0\,\right)\left(\,0 \quad 10^{-6}\,\right)\right)$.

```
ClearAll[update];
update[{ψ_, Λ_}, {ξ_, a_}] :=
  With[{Π = (Λ + aᵀ.a)},
   {Inverse[Π].(aᵀ.ξ + Λ.ψ), Π}];
MatrixForm /@
 ({⎛mBar⎞, Π} = Fold[update,
   ⎝bBar⎠
    {⎛0⎞, ⎛1.0*^-6     0    ⎞},
     ⎝0⎠  ⎝   0     1.0*^-6 ⎠
    {List /@ data, List /@ partials}ᵀ])
```

```
{⎛ 0.568383 ⎞, ⎛280.356  119.⎞}
 ⎝-0.503288 ⎠  ⎝ 119.    119.⎠
```

The estimates **mBar** and **bBar** are exactly what we got from Wolfram’s built-in.

The mappings of **List** over the data and partials convert them into column vectors, as required by the recurrences in linear-algebra form.

The final value of Λ (called Π in the code, a returned value), is $A_{\mathrm{full}}^{\mathsf{T}} \cdot A_{\mathrm{full}} + \Lambda_0$:

```
Π - partialsᵀ.partials
```

```
{{1. × 10^-6, 0.}, {0., 1. × 10^-6}}
```

The covariance of the estimate Ξ is $\left(\frac{n-1}{n-2}\right)*$Variance[Z – $A \cdot \Xi$]$*\Lambda^{-1}$ except for a small contribution from the a-priori covariance Λ_0. The correction factor $\left(\frac{n-1}{n-2}\right)$ is a generalization of Bessel’s correction. The 2 in $(n-2)$ in the denominator is due to the fact that we’re estimating two parameters from the data (see VAN DE GEER, Least Squares Estimation, Volume 2, pp. 1041–1045, in Encyclopedia of Statistics in Behavioral Science, Eds. Brian S. Everitt & David C. Howell, Wiley, 2005). The denominator of the correction, in general, is $n - p$, where n is the number of data and p is the number of parameters being estimated.

```
                                  nData - 1
Inverse[partialsᵀ.partials] * ─────────── *
                                  nData - 2
  Variance[data - partials.{mBar, bBar}] // MatrixForm
```

```
⎛ 0.00259377  -0.00259377⎞
⎝-0.00259377   0.00611074⎠
```

Except for the reversed order, this is the same covariance matrix that Wolfram reports:

```
Reverse@(Reverse /@ model["CovarianceMatrix"]) // MatrixForm
```

```
( 0.00259377  -0.00259377 )
( -0.00259377  0.00611074 )
```

Don’t Invert That Matrix

See https://www.johndcook.com/blog/2010/01/19/dont-invert-that-matrix/

Replace **Inverse** with **LinearSolve**:

```
ClearAll[update];
update[{ψ_, Λ_}, {ζ_, a_}] :=
  With[{Π = (Λ + aᵀ.a)},
   {LinearSolve[Π, (aᵀ.ζ + Λ.ψ)], Π}];
MatrixForm /@ ({( mBar ), Π} = Fold[update,
                ( bBar )
    {( 0 ), ( 1.0*^-6     0     )},
     ( 0 )  (    0     1.0*^-6  )
    {List /@ data, List /@ partials}ᵀ])
```

```
{( 0.568383  ), ( 280.356  119. )}
 ( -0.503288 )  (  119.    119. )
```

exactly as before.

We have eliminated memory bloat by accumulating estimate updates one observation at a time, each with its paired partial. We reduce computation time and numerical risk by solving a linear system instead of inverting a matrix.

The Dual Problem

When the model is a co-vector (row-vector), e.g., a gradient, we have the dual (transposed) problem. In that case, the observations Ω and the model Γ are now co-vectors with elements ω and γ instead of ζ and ψ. The co-partials Θ (replacing A) are now a co-vector of vectors θ. The observation equation looks like $\Omega = \Gamma \cdot \Theta$ and the error-so-far like $(x - \gamma) \cdot \Lambda \cdot (x - \gamma)^{\intercal}$, where $\Lambda = \Theta_{\text{so-far}} \cdot \Theta_{\text{so-far}}{}^{\intercal}$. We don’t change the name of Λ because it’s symmetric. Adding a new observation ω introduces new error $(\omega - x \cdot \theta) \cdot (\omega - x \cdot \theta)^{\intercal}$. Minimizing the total error yields

$$\begin{aligned} \gamma &\leftarrow (\gamma \cdot \Lambda + \omega \cdot \theta^{\intercal}).(\Lambda + \theta \cdot \theta^{\intercal})^{-1} \\ \Lambda &\leftarrow (\Lambda + \theta \cdot \theta^{\intercal}) \end{aligned} \tag{4}$$

LinearSolve operates on the transposed right-hand side of the recurrence, and we transpose the solution to get the recurrence.

```
Transpose /@ List /@ partials⟦1 ;; 3⟧
```

```
{{{-1.}, {1.}}, {{-0.966102}, {1.}}, {{-0.932203}, {1.}}}
```

```
ClearAll[coUpdate];
coUpdate[{γ_, Λ_}, {ω_, θ_}] :=
  With[{Π = (Λ + θ.θᵀ)},
   {LinearSolve[Π, Λ.γᵀ + θ.ωᵀ]ᵀ, Π}];
MatrixForm /@ Fold[
  coUpdate,
  {( 0  0 ), (1.0*^-6     0    )},
             (   0     1.0*^-6 )
  {List /@ List /@ data, Transpose /@ List /@ partials}ᵀ]
```

```
{( 0.568383  -0.503288 ), (280.356  119.)}
                          ( 119.    119.)
```

Application of the Dual Problem

Imagine a scalar function $J(\theta)$ of a column K-vector $\theta_{K\times1}$. We want to estimate its gradient co-vector ∇J, given a batch of $\mathcal{I}$ random increments $\Delta\Theta_{K\times\mathcal{I}}$, from the system $\nabla J\cdot\Delta\Theta=\Delta J$. Here, ∇J takes the role of the model whose state parameters Γ we want to estimate, $\Delta\Theta$ takes the role of the partials of the model w.r.t. those parameters, and ΔJ takes the role of measured data. Let $\Delta J_{\mathcal{I}\times1}$ be a *batch* of observed increments to J and $\Delta\Theta_{K\times\mathcal{I}}$ be a matrix of the $\mathcal{I}$ corresponding column-vector random increments to the input vectors $\theta_{K\times1}$. The right pseudoinverse $\mathrm{RPI}\overset{\mathrm{def}}{=}(\Delta\Theta_{K\times\mathcal{I}}\cdot\Delta\Theta^{\intercal}_{K\times\mathcal{I}})^{-1}\cdot\Delta\Theta^{\intercal}_{K\times\mathcal{I}}$ solves $\nabla J_{\mathcal{I}\times1}\cdot\Delta\Theta_{K\times\mathcal{I}}=\Delta J_{\mathcal{I}\times1}$ to yield $\nabla J_{\mathcal{I}\times1}\approx\Delta J_{\mathcal{I}\times1}\cdot\mathrm{RPI}$. Use the co-update recurrence for this problem.