DATA 1010
IN-CLASS EXERCISES
SAMUEL S. WATSON
14 SEPTEMBER 2018

Problem 1

Suppose that A is an $m \times n$ matrix, that $\mathbf{b} \in \mathbb{R}^m$, and that $\lambda > 0$. Find a formula for the value of $\mathbf{x}$ which minimizes

$$|A\mathbf{x} - \mathbf{b}|^2 + \lambda|\mathbf{x}|^2.$$

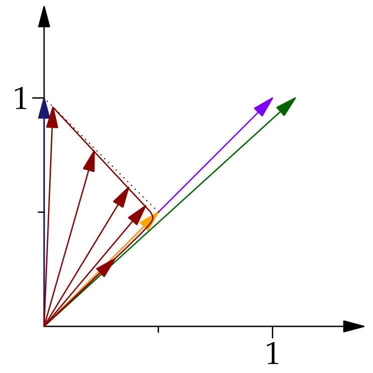
(For any square matrices that you would like to invert, you may assume they are invertible). Describe what happens (either to the minimizer, or to the original optimization problem) as λ increases from 0.

Solution (and broader discussion)

Suppose that A is an $m \times n$ matrix and $\mathbf{b}$ is an m by 1 column vector.

The least squares problem asks to us find the linear combination of the columns of A which is closest to b. However, sometimes that isn't really what we want, because the weights of the linear combination might be very large.

For example, suppose we take a nearly singular matrix like $\begin{bmatrix} 1.1 & 1 \\ 1 & 1 \end{bmatrix}$ and ask what linear combinations of the columns is closest to $[0, 1]$. Then we can get $\mathbf{b}$ exactly: we need -10 of the first column and 11 of the second column. If we replace 1.1 with numbers increasingly close to 1, then the weights would go off to $-\infty$ and ∞, respectively. These large weights reflect an effort to leverage the small difference between the two columns to "reach out" in the direction of $[0, 1]$. If we don't want such small differences to have such a big impact on the solution, then we might be happier to choose a point more like $[1/2, 1/2]$, which is the point closest to $\mathbf{b}$ among those in the $[1, 1]$ direction (which is the essentially the direction represented by the vectors $[1, 1]$ and $[1.1, 1]$).



One way to formalize this idea is to add a term in the function we're trying to minimize which penalizes large weights. We include a parameter λ which we can tune to increase or decrease this penalty.

To find where this function has a minimum, we can differentiate it with respect to $\mathbf{x}$ and set the resulting expression equal to zero. We find that

$$\begin{aligned} \frac{\partial}{\partial \mathbf{x}} \left((A\mathbf{x} - \mathbf{b})'(A\mathbf{x} - \mathbf{b}) + \lambda \mathbf{x}'\mathbf{x} \right) &= (A\mathbf{x} - \mathbf{b})'A + (A\mathbf{x} - \mathbf{b})'A + \lambda \mathbf{x}' \frac{\partial}{\partial \mathbf{x}}\mathbf{x} + \lambda \mathbf{x}' \frac{\partial}{\partial \mathbf{x}}\mathbf{x} \quad \text{(product rule)} \\ &= 2(A\mathbf{x} - \mathbf{b})'A + 2\lambda \mathbf{x}' \\ &= 2(\mathbf{x}'A' - \mathbf{b}')A + 2\lambda \mathbf{x}' \\ &= 2(\mathbf{x}'(A'A + \lambda I) - \mathbf{b}A'). \end{aligned}$$

Setting this equal to zero and taking the transpose of both sides, we find that $\mathbf{x}$ is equal to $(A'A + \lambda I)^{-1}A'\mathbf{b}$.

Thus the usual least squares formula is modified by adding a multiple of the identity to the $A'A$ term. Since that term is being inverted, we can see how increasing the value of λ would generally decrease the entries in the solution.

Testing this out computationally, we find that for small values of λ, we get a vector which points approximately due north, while for large values of λ, we get a vector which points more in the $[1,1]$ direction but is very short (see the red vectors in the image). In between, there is a value of λ which gives us a vector very close to the $[1/2, 1/2]$ vector we were looking for.

Another approach which gives a similar result in this case would be to find the singular value decomposition $U\Sigma V'$ of A and project $\mathbf{b}$ onto the span of the columns of U which correspond to non-small singular values. In this case, that procedure results in $[0.499, 0.475]$.

Problem 2

Show that $x - y = 0$ if and only if $x = y$ (where the subtraction is a `Float64` operation). Show that this is *not* the case in the variant of the `Float64` system which lacks subnormal numbers.

Solution

The difference between two positive representable numbers is always 2^k where k is between -1074 and 971. We can represent every such number, so the difference between two different numbers never rounds to zero.

If we excluded the subnormal numbers, then there would many differences between distinct numbers which would round to zero. For example, the difference between 2^{-1022} and the number one tick to the right, which is $2^{-1022} + 2^{-1074}$, would be much smaller than the smallest positive representable number. Therefore, this difference would round to zero.

Problem 3

Select the numbers which are exactly representable as a `Float64`.

$$2^{1024} \quad \frac{4}{3} \quad -2^{-1074} \quad \frac{3}{8} \quad 0.0 \quad 1.5 \quad 0.8$$

Solution

A positive number is representable as a normal `Float64` value if and only if exceeds some power of 2 between 2^{-1022} and 2^{1023} by a multiple of that power of 2 times $(1/2)^{52}$. Therefore, every modest-sized rational number whose denominator is a relatively small power of 2 is representable, and every rational number with denominator *not* equal to a power of 2 is not representable.

2^{1024} is the first power of 2 which is not representable.

$4/3$ is not representable since it is reduced, but its denominator is not a power of 2.

2^{-1074} is representable (it's the least representable negative number).

$3/8$ is representable since it exceeds $1/4$ by a small multiple of a power of $1/2$.

0.0 is representable as a subnormal number.

1.5 is representable since it's $1/2$ more than 1. 0.8 is equal to $4/5$, so it is not representable.

Problem 4

In Julia, the function `nextfloat` returns the next largest representable value. Predict the values returned by the following lines of code, and then run them to confirm your predictions.

```
log2(nextfloat(15.0)-15.0)
log2(nextfloat(0.0))
log2(1.0 - prevfloat(1.0))
```

Solution

The difference between 15.0 and the next representable float is the tick spacing for the interval from 8 to 16. From [1,2) to [8,16), the tick spacing doubles three times, so the tick spacing in that interval is 2^{-49}, so the first line returns -49.

The least positive representable float is 2^{-1074}, so the second line should return -1074.

The difference between 1.0 and the next float down is the tick spacing in the interval from $[1/2 to 1)$. This is half the tick spacing in the interval from 1 to 2, so this lines returns -53.

Problem 5

Investigate the rounding behavior when the result of a calculation is exactly between two representable values. Define `ϵ = 1/2^52` and check whether `1.0 + 0.5ϵ == 1.0`. Repeat with 1.5 in place of 0.5 (and appropriate changes made to the right-hand side). What does the rounding rule appear to be?

Solution

Adding a half tick spacing to 1 results in 1. So we can see that the point in the middle of that interval gets rounded down. However, adding a tick spacing and a half results in 1 plus *two* tick spacings. In fact, midpoints get rounded to the nearest *even*-numbered tick in the relevant interval between successive powers of 2 (where the tick count in that interval starts at 0).

Bonus

Calculating inverse square roots is a very common task in graphics-intensive settings like video games. In the late 1990's, the following algorithm for approximating the inverse square root function appeared in the source code of the game *Quake III Arena*. The operator `>>` shifts the bits in the underlying representation over by 1 position, and `reinterpret` creates a new instance of the given type whose bits are the same as the bits of the given value.

```
function invsquareroot(x::Float32)
    y = 0x5f3759df - (reinterpret(Int32,x) >> 1)
    z = reinterpret(Float32,y)
    z * (1.5f0 - (0.5f0*x)*z*z)
end
```

If you are amazed by the appearance of the magic constant `0x5f3759df`* so was the original author. You can see their code comments on the Wikipedia entry for Fast Inverse Square Root.) Evaluate this function with a few input values and determine its relative error on each. Define another function which calculates the inverse square root in the obvious way (`1/sqrt(x)`) and check that the one above actually does run faster. As a double extra bonus, figure out why this code works.

this is syntax for an unsigned, 32-bit integer