Multidimensional Data Analysis

Scientific Computing in Rust, 2024

Libor Spacek
(liborty@github.com)
https://github.com/liborty

18th July 2024

Overview

Introduction

- ‘Scientific Computing’ implies working with numbers
- Primary data structures used: Vec$<$T$>$, Vec$<$ Vec$<$T$>$ $>$ and their slices
- Using Vec, we were able to combine in one crate: Statistics, Information Theory, Vector Algebra (operations on several vectors), Linear Algebra (matrices) and Data Analysis (many multi-dimensional *nd* vectors)
- Our treatment of *nd* data is constructed from the first principles. Some novel concepts are introduced and implemented

Test Problem

Numer.ai tournament competition provides a lot of numbers:

- Five ordered outcome classes (0.0, 0.25, 0.5, 0.75, 1.0)
- Current training set of 2,790,013 instances (changing weekly)
- Daily set of 4,917 instances, each to be classified by a single number in (0,1). Any placed out of order incur penalties
- Each instance has 2,376 features. It is one point in 2,376 dimensional space. Outcome classes form large, densely intersecting clouds (not all of the same size)
- Most data analysis and machine learning (ML) problems can be formulated in these terms, though not readily 'solved'. Instead, many people resort to 'black box' of a neural network.

Dimensions Reduction

- In the forlorn hope of reducing the data and gaining more focus, some attempt PCA. However, this adds significant load with iterative eigenvalues computation in the original large space
- We select significant axis based on small values of their Mahalanobis scaling. This is more manageable, as it only requires one efficient Cholesky matrix decomposition
- Each cloud is processed individually. Each then has its own subspace based on its own shape, which aids classification.

Reflections on Rust

- Implemented a number of related crates on crates.io: **ran** (random numbers), **times** (benchmarking), **medians** (in 1d), **indxvec** (sorting, searching, indexing, printing), **sets**, **rstats**
- Rstats is the main crate for the purposes of this presentation
- + Rust allows running the above large problem on my home desktop (impractical with Python)
- + Functional chaining, + no execution errors
- + Easy multi threading (with rayon)
- - Implementing generic traits for Vec should be easier.

Implementation

The main constituent parts of Rstats are its generic traits. All data are Vecs of arbitrary length d (dimensionality). The traits are mostly distinguished by the number of Vec arguments their methods take:

- `Stats`: a single collection of numbers (1 argument)
- `Vecg`: methods of vector algebra and information theory (2 arguments, e.g. scalar product)
- `MutVecg`: some of the above methods, mutating self
- `Vecu8`: some methods implemented more efficiently for u8
- `VecVec`: 'self' is vector of vectors: n vectors in d dimensions
- `VecVecg`: takes an extra generic argument, typically a vector of weights. For example, to find weighted geometric median of points with varying importance (such as time dependence).

Some New Concepts

Geometric median is stable and reduces the undue influence of outliers. Thus zero median vectors are generally preferred to the commonly used zero mean vectors

- `median correlation` - we normalise both data samples to their zero median forms (instead of Pearson's zero mean form). Treating them as vectors, we define the median correlation as cosine of an angle between them (same as Pearson)
- `comediance matrix (nd)` - like covariance matrix but computed from zero median data, obtained by setting the origin to the geometric median.
- `madgm (nd)` - generalisation of robust data spread estimator known as 'MAD': median of absolute deviations from median (1d). In nd, we replace the deviations by the distances from the geometric median (already always positive).

Code Example and Benchmark Test

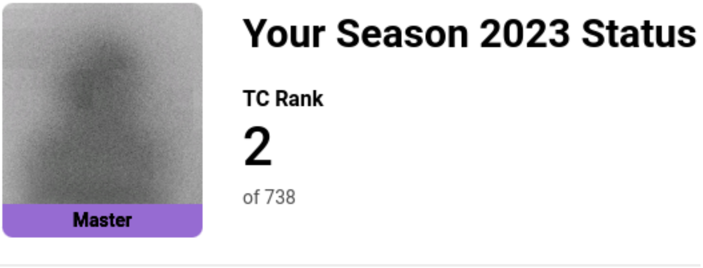
- Arithmetic *nd* mean is where the sum of vectors is zero. Geometric median (gm) is where the sum of *unit* vectors is zero (less susceptible to outliers but can only be found iteratively)
- My gm algorithm - perhaps not the fastest possible but relatively simple and easy to parallelise
- Solves the instability and convergence problems of the original Weiszfeld algorithm
- The benchmark comparison deploys my crate **times**

```rust
fn gmedian(self, eps: f64) -> Vec<f64> {
    let mut g = self.acentroid(); // start iterating from the mean  or vec![0_f64; self[0].len()];
    let mut recsum = 0f64;
    loop {
        // vector iteration till accuracy eps is exceeded
        let mut nextg = vec![0_f64; self[0].len()];
        let mut nextrecsum = 0_f64;
        for p in self {
            // |p-g| done in-place for speed. Could have simply called p.vdist(g)
            let mag: f64 = p &Vec<T>
                .iter() Iter<T>
                .zip(&g) impl Iterator<Item = (&T, &f64)>
                .map(|(vi, gi)| (vi.clone().into() - gi).powi(2)) Map<Zip<Iter<T>, Iter<f64>>, …>
                .sum();
            if mag > eps {
                // reciprocal of distance (scalar)
                let rec = 1.0_f64 / (mag.sqrt());
                // vsum increment by components
                for (vi, gi) in p.iter().zip(&mut nextg) {
                    *gi += vi.clone().into() * rec
                }
                // add the scaling reciprocal
                nextrecsum += rec
            } // ignore point p when |p-g| <= eps
        }
        nextg.iter_mut().for_each(|gi| *gi /= nextrecsum);
        if nextrecsum - recsum < eps {
            return nextg;
        }; // termination
        g = nextg;
        recsum = nextrecsum;
    }
} fn gmedian
```

```
Timing Comparisons (in nanoseconds):

Data:&[Vec<f64>] lengths:1000-1500 step:200 rows:100 repeats:10

Length: 1000

par_acentroid        1269992 ±  86328 ~ 6.80%  1.0000
acentroid            1338256 ±   2655 ~ 0.20%  1.0538
par_gmedian          3431354 ± 358287 ~10.44%  2.7019
gmedian              8510493 ±  67658 ~ 0.79%  6.7012
quasimedian         12147504 ±  73984 ~ 0.61%  9.5650

Length: 1200

par_acentroid        1407743 ±  34941 ~ 2.48%  1.0000
acentroid            1682506 ±   3401 ~ 0.20%  1.1952
par_gmedian          4335982 ± 187800 ~ 4.33%  3.0801
gmedian             10620628 ±  25794 ~ 0.24%  7.5444
quasimedian         15356484 ±  34896 ~ 0.23% 10.9086

Length: 1400

par_acentroid        1624728 ±  54423 ~ 3.35%  1.0000
acentroid            1969138 ±   8221 ~ 0.42%  1.2120
par_gmedian          5082391 ± 157718 ~ 3.10%  3.1281
gmedian             12425460 ±  87615 ~ 0.71%  7.6477
quasimedian         18022492 ± 120432 ~ 0.67% 11.0926

Total errors for 10 repeats of 100 points in 1000 dimensions:

par_gmedian   0.0001911485
gmedian       0.0001911485
acentroid     1.3061089217
par_acentroid 1.3061089217
quasimedian   87.9149666326
```

Conclusion

How does this approach compete with 738 data scientists, running existing neural nets libraries in SciPy?



(I messed up with their new data format and got pipped to the post)