incr_stats Public

main 1 branch 0 tags Go to file Add file Code

GaryBoone version bump 71c233b on Feb 16 29 commits

benches	added update_bench; README revisions	9 months ago
examples	Updated benchmarks and README	9 months ago
src	comment corrections/clarifications	9 months ago
.gitignore	comment corrections/clarifications	9 months ago
Cargo.toml	version bump	9 months ago
LICENSE-APACHE	comment corrections/clarifications	9 months ago
LICENSE-MIT	comment corrections/clarifications	9 months ago
README.md	description tweaks	9 months ago

incr_stats

Fast, Scalable, Incremental Descriptive Statistics in Rust

`incr_stats` provides time- and memory-optimized, scalable, descriptive statistics with the following features:

- Incremental (aka `streaming`) updates mean that no source data is stored
- Calculation of the stats at request time is nearly instantaneous
- Faster than non-incremental equivalents, while requiring far less storage
- Allows fast update of already-calculated statistics
- Especially appropriate for low-memory applications or applications where the dataset is too large to store such as continuous data generation
- Provides descriptive statistics including skewness and kurtosis.
- Both population and sample statistics
- Protection against error-inducing NaNs and Infs
- Extensive tests for accuracy
- Validated against R and Octave
- Additional optimized batch functions provided for each statistic
- Equivalent textbook batch functions provided for each statistic
- Speed benchmarks compare incremental and batch calculations

Statistics Included

The `incr_stats` Stats package provides the following descriptive statistics:

- `count()`
- `min()`
- `max()`
- `sum()`
- `mean()`

About

Fast, scalable, incremental descriptive statistics in Rust

Readme
Apache-2.0, MIT licenses found
Activity
2 stars
2 watching
0 forks
Report repository

Releases

No releases published

Packages

No packages published

Languages

Rust 100.0%

- `population_variance()`
- `sample_variance()`
- `population_standard deviation()`
- `sample_standard deviation()`
- `population_skewness()`
- `sample_skewness()`
- `population_kurtosis()`
- `sample_kurtosis()`

Examples

The `incr_stats` Stats package operates on `f64` data and is easy to use.

Incremental/Streaming

It's not necessary to store the entire data stream to calculate its descriptive statistics.

```
use incr_stats::incr::Stats;

let mut s = Stats::new();

// Update the stats as data becomes available, without storing it.
s.update(1.2)?;
s.update(0.2)?;
// ...

// Calculate the descriptive statistics as needed.
println!("The skewness is {:.4}", s.sample_skewness()?);
println!("The kurtosis is {:.4}", s.sample_kurtosis()?);
```

Some calculations are done by each `update()`, so they are slower than simply storing the value. However, very little calculation then need be done to generate the requested statistic, so `sample_kurtosis()`, for example, is nearly instantaneous compared to the traditional algorithms that operate on the entire array.

An `array_update()` function is also provided so that incremental updates can be performed on a group of values.

The `incr_stats` crate contains two other versions of the same calculations for comparison. In general, the incremental version is the fastest, but they all produce identical results.

This example is in `examples/incr_example.ps` and can be run with `$ cargo run --example incr_example`.

Memoized

The `vec` version requires stored data, but is optimized and provides the same accuracy. Descriptive statistics depend on each other, such as the skewness depending on the variance which depends on the mean. This version ensures that only the minimal calculations are performed for the set of statistics, no matter which statistic is requested. Further, subsequent requests do not repeat already-done calculations.

```
use incr_stats::vec::Stats;

let a = vec![1.2, -1.0, 2.3, 10.0, -3.0, 3.2, 0.33, 0.23, 0.23, 1.0];
let mut s = Stats::new(&a)?;

println!("The skewness is {:.4}", d.sample_skewness()?);
println!("The kurtosis is {:.4}", d.sample_kurtosis()?);
```

This example is in `examples/vec_example.ps` and can be run with `$ cargo run --example vec_example`.

Batch

Finally, a third version uses traditional, textbook calculations. These do the required calculations with no other overhead. They are included primarily for comparison and testing, but can be fastest for stored data if only one or two statistics is needed.

```
use incr_stats::batch;

let a = vec![1.2, -1.0, 2.3, 10.0, -3.0, 3.2, 0.33, 0.23, 0.23, 1.0];

println!("The skewness is {:.4}", batch::sample_skewness(&a)?);
println!("The kurtosis is {:.4}", batch::sample_kurtosis(&a)?);
```

This example is in `examples/batch_example.ps` and can be run with `$ cargo run --example batch_example`.

Which to use?

Choose `incr` stats first, unless your use fits an optimization described below.

`incr`, the incremental stats:

The `incr` stats does not store the data, but instead stores only a few intermediate values needed to calculate the complete statistics when one is requested. It is updated one or a few values at a time.

The incremental stats are ultimately equivalent in calculation to the batch calculations, except that the calculations are amortized over the individual data updates. It's fastest way to calculate the descriptive statistics on large datasets.

It's also appropriate if:

- memory is limited
- the data is unlimited, such as in streaming or continuous data applications
- you don't want to store and manage all of the data
- the current stats need to be updated
- data becomes available only one or a few points at a time, but overall stats are needed
- calculation of the stats, when requested, must be fast, ie there isn't time to calculate over the entire dataset.

`vec`, the optimized vector stats:

The tradeoff for eliminating data storage and making the final statistical calculations fast is that performing the intermediate calculations for each `update()` is slower than simply storing the data.

The `vec` stats struct memoizes intermediate results while performing calculations on stored data. This optimization ensures that dependent calculations such as the mean and variance are reused when needed by other stats. Repeated calls become look-ups instead of recalculations.

Use the `vec` stats if:

- the application cannot provide enough time for the incremental `update()` but only has time to store the data
- you're only interested in one or two statistics, so the overhead of the `update()` calculations is wasted on statistics that won't be requested
- it's ok if the calculations of the final statistics is slow due to operating on the entire dataset.

`batch`, the unoptimized textbook equations:

The `batch` functions are mainly included for accuracy comparisons and testing, but they can be slightly faster than the `vec` version which has some overhead due to checking for previously-calculated values. If only one statistic is needed, then there's no reuse among statistics that would make the `vec` version faster.

Use the `batch` stats if:

- the absolutely fastest calculation of a *single* statistic (eg, `sample_kurtosis`) is required

Population and Sample Statistics

`incr_stats` provides both population and sample statistics.

The code documents the corresponding functions in the R and GNU Octave stats packages, clarifying their naming and parameter differences.

Accuracy: R and Octave Validation

`incr_stats` unit tests include examples that are confirmed to match results of the R and GNU Octave stats packages to 13 decimal places.

See the code for the corresponding R and GNU Octave functions.

Speed

README.md

when a statistics is requested. In contrast, incremental stats do some processing at acquisition, allowing the final statistics calculations to be very fast.

So with respect to speed, it's really a question of where you want to spend the calculation time:

- A little at a time as the data is collected? Then updates are slow but the final stats are fast.
- All at the end when a statistic is requested? Then updates are fast, just storing the data, but the final stats are slow.

The incremental statistics carry the overhead of doing the calculations of statistical moments for each data point update. The overhead allows the complete descriptive statistics to be calculated quickly at any time without storing the entire dataset. It may appear to be a considerable amount of calculation for just one data point. And it may appear that a lot more calculation is done over all of the points than is done in the stored-array cases. But in fact nearly identical calculations must be done for the stored-array versions, so the two are nearly identical in terms of processing time, overall. The stored-array versions loop through the data several times; the incremental versions unroll these loops, splitting each loop calculation into a separate `update()` performed when the value is acquired.

This is why the only reason to prefer stored-array statistics is because:

1. the data must be processed as quickly as possible at acquisition; there's only time to store it, or
2. you only need one or two of the statistics, so for example there's no need to calculate the intermediate values for the 4th moment at each `update()` when kurtosis will not be requested.

Benchmarks

Criterion benchmarks are included to allow comparisons of the incremental, memoized, and batch statistics calculations. The actual experimental times will vary between machines and operating systems, so here we consider these representative and make relative comparisons.

For datasets with 10 and 1,000,000 randomized values, here is the total time to calculate [lower is better]:

`Kurtosis` means that only the `sample_kurtosis` was calculated, showing the time if only one statistic is needed. `All stats` means that all of the statistics (`count`, `min/max`, `sum`, `mean`, `{samp/pop}_variance`, `{samp/pop}_standard_deviation`, `{samp/pop}_skewness`, `{samp/pop}_kurtosis`) were calculated.

Method	10, Kurtosis	10, All stats	1M, Kurtosis	1M, All stats
incr	81.736 ns	101.35 ns	7.004 ms	7.105 ms
batch	40.124 ns	228.20 ns	3.830 ms	29.128 ms
vec	68.094 ns	106.80 ns	4.124 ms	8.411 ms

Analysis

When multiple statistics are needed, the incremental (`incr`) stats is fastest, regardless of dataset size.

As mentioned above, the amount of calculation is similar in both the incremental and the optimized stored-array (`vec`) versions, but the incremental version is faster than the optimized stored-array version by 5.1% (10 data points) to 15.5% (1M data points).

As expected, for calculating just one statistic, the unoptimized stored-array version (`batch`) code is fastest, independent of dataset size. That's because it performs the minimal calculations for just one statistic.

Note that for the incremental version on large (1M) datasets, the time required for calculating all of the stats (`7.105 ms`) was very close (1.4%) to calculating just the sample kurtosis (`7.004 ms`) in this benchmark run. That's because for the incremental version, the majority of the effort is spent on the individual point update calculations which includes intermediate values needed for all of the statistics. As a result, the times required for calculating the final `all stats` versus any individual statistic converge.

`incr` update and final times

The incremental statistic package spends time on every `update()` doing some calculations. How expensive are these? And how fast is final calculation time for `incr` ?

Method	10 values	1M values
update	11.722 ns	11.843 ns
final	2.591 ns	2.584 ns

This table shows the times for each individual `update()` and the final calculation of `sample_kurtosis()` for 10 and 1 million values.

As expected, there's no significant difference in times due to dataset size.

We see that each update requires <12ns to calculate the intermediate moments. This work, spread over all of the updates, enables the final calculation of `sample_kurtosis()` to need only 2.6ns, roughly a fifth of the update time.

Error Handling

The `incr_stats` crate handles errors in a simple and consistent way. There are only three kinds of errors:

1. `NotEnoughData` : This error merely means that more data is needed to allow the calculation of the statistic. For example, the sample skewness calculation

includes a division by `n-1` so must include at least 2 data points to avoid a division by 0.0.

2. `Undefined` : Even with enough valid data, some statistics produce undefined results. For example, if all of the data is the same value, the variance is 0.0. Skewness and kurtosis, which are measures of the distribution around a central tendency, don't exist. This fact is reflected in the calculations by a division by the variance (ie a divide by 0.0). These are therefore undefined.
3. `InvalidData` : The floating data is checked for NaNs and Infs from the `IEEE 754` standard.

Callers that don't need to make these distinctions can just react to any error.

License