fn score_candidates

Michael Shoemate

October 2, 2025

This proof resides in **"contrib"** because it has not completed the vetting process.

Proves soundness of `score_candidates` in `mod.rs` at commit f5bb719 (outdated[1]). `score_candidates` returns a score for each candidate passed in, where the score is the distance between the candidate and the ideal alpha-quantile.

1 Hoare Triple

Precondition

Compiler Verified

- Generic `TI` (input atom type) is a type with trait `PartialOrd`.

Caller Verified

- `x` has at most 2^{64} elements.
- `x` elements are totally ordered (excluding non-nan/non-null).
- `candidates` is strictly increasing
- `alpha_numer / alpha_denom` ≤ 1
- `size_limit` $\cdot$ `alpha_denom < u64.MAX`

Function

```python
def score_candidates(
    x: Iterator[TIA],
    candidates: list[TIA],
    alpha_num: u64,
    alpha_den: u64,
    size_limit: u64
) -> Iterator[usize]:
    # count of the number of records between...
    #  (-inf, c1), [c1, c2), [c2, c3), ..., [ck, inf)
    hist_ro = [0] * candidates.len() + 1 # histogram of right-open intervals
    #  (-inf, c1], (c1, c2], (c2, c3], ..., (ck, inf)
    hist_lo = [0] * candidates.len() + 1 # histogram of left-open intervals

    for x_i in x:
```

[1]See new changes with `git diff f5bb719..bf9b9f8 rust/src/transformations/quantile_score_candidates/mod.rs`

```
15          idx_lt = candidates.partition_point(lambda c: c < x_i)
16          hist_lo[idx_lt] += 1  #
17
18          idx_eq = idx_lt + candidates[idx_lt:].partition_point(lambda c: c == x_i)
19          hist_ro[idx_eq] += 1  #
20
21      n: u64 = hist_lo.iter().sum()  #
22
23      # don't care about the number of elements greater than all candidates
24      hist_ro.pop()  #
25      hist_lo.pop()  #
26
27      lt, le = 0, 0
28      for ro, lo in zip(hist_ro, hist_lo):  #
29          # cumsum the right-open histogram to get the total number of records less than the candidate
30          lt += ro  #
31          # cumsum the right-open histogram to get the total number of records lt or equal to the candidate
32          le += lo  #
33
34          gt = n - le #
35
36          # the number of records equal to the candidate is the difference between the two cumsums
37          lt_lim, gt_lim = lt.min(size_limit), gt.min(size_limit)  #
38
39          # a_den * |    (1 - a)         * #(x < c)     -             a * #(x > c)|
40          yield ((alpha_den - alpha_num) * lt_lim).abs_diff(alpha_num * gt_lim)  #
```

Postcondition

Theorem 1.1. Let C denote `candidates` and let l denote `size_limit`. For each index i in $\{1, \ldots, |C|\}$,

$$\text{score\_candidates}_i(x, C, \alpha_{num}, \alpha_{den}, l) = |\alpha_{den} \cdot \min(\#(x < C_i), l) - \alpha_{num} \cdot \min(\#(x > C_i), l)|$$

where $\#(x < C_i)$ is the number of elements in x less than C_i, $\#(x > C_i)$ is the number of elements in x greater than C_i.

Proof. The function breaks down into two parts:

- Compute histograms, where the edges are the candidates.
- Uses the histograms to compute scores.

The histograms are initialized at zero, with one more bin than candidates, since the bins start at $-\infty$ and end at $+\infty$.

The bins in `hist_ro` are closed on the left, and open on the right. The bins in `hist_lo` are open on the left, and closed on the right.

This is reflected in line 16, where `idx_lt` is the index of the first bin smaller than `x_i`. Similarly, `idx_eq` is the index of the first bin greater than or equal to `x_i`. It is sufficient to search on equality, because if no candidate is equal to `x_i`, then the partition point will be zero, so `idx_eq` will be equal to `idx_lt`, as expected.

Line 21 computes the total number of elements in the data as each element in `x` incremented a bin in `hist_lo`. No arithmetic thus far can overflow due to the precondition that `x` has at most 2^64 elements.

Notice that there is one more bin than candidates. With `n`, the last bin in each of the histograms (the number of elements beyond the largest candidate), is not needed to compute the score. Therefore, the last bin is discarded on lines 24 and 25.

The scores can now be computed in one linear pass over the histograms on line 28. The number of elements less than the candidate `lt` is the cumulative sum of `hist_lo`, and the number of elements less than or equal to the candidate `le` is the cumulative sum of `hist_ro`.

The number of elements greater than the candidate `gt` is then simply `n - le` on line 34.

To ensure that the score computation does not overflow, the counts are bounded by `size_limit` on line 37.

By the precondition that `alpha_numer / alpha_denom` ≤ 1, then both `alpha_den - alpha_num` and `alpha_num` are less than or equal to `alpha_den`. Using this, and the precondition that `size_limit` $\cdot$ `alpha_denom < u64.MAX`, then the computation of the score on line 40 is guaranteed to not overflow.

The computation on line 40 directly satisfies the postcondition. □