

fn compute_score

Michael Shoemate

June 20, 2024

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

Proves soundness of `compute_score` in `mod.rs` at commit `f5bb719` (outdated¹). `compute_score` returns a score for each candidate passed in, where the score is the distance between the candidate and the ideal alpha-quantile.

Vetting History

- [Pull Request #456](#)

1 Hoare Triple

Precondition

- TIA (input atom type) is a type with trait `PartialOrd`.
- `x` is non-null
- `candidates` is strictly increasing
- `alpha_numer / alpha_denom` is in $[0, 1]$
- `size_limit * alpha_denom` does not overflow

Function

```
1 def compute_score(  
2     x: list[TIA],  
3     candidates: list[TIA],  
4     alpha_num: usize,  
5     alpha_den: usize,  
6     size_limit: usize  
7 ) -> list[usize]:  
8  
9     x = list(sorted(x))  
10  
11     num_lt = [0] * len(candidates)  
12     num_eq = [0] * len(candidates)  
13  
14     count_lt_eq_recursive(  
15         num_lt, # mutated in-place  
16         num_eq, # mutated in-place
```

¹See new changes with `git diff f5bb719..d7c4dce rust/src/transformations/quantile_score_candidates/mod.rs`

```

17     edges=candidates,
18     x=x,
19     x_start_idx=0)
20
21     def score(lt, eq):
22         return abs_diff(
23             alpha_den * min(lt, size_limit),
24             alpha_num * min(len(x) - eq, size_limit))
25
26     return [score(lt, eq) for lt, eq in zip(num_lt, num_eq)]

```

Postcondition

Each element in the return value corresponds to the score of the candidate at the same index in `candidates`:

$$\text{compute_score}(X, c, \alpha_{num}, \alpha_{den}, l) = |\alpha_{den} \cdot \min(\#(X < c), l), \alpha_{num} \cdot \min(|X| - \#(X = c), l)| \quad (1)$$

2 Proof

By the preconditions on `compute_count`, and by sorting `x`, the preconditions on `count_lt_eq_recursive` are satisfied.

By the definition of `count_lt_eq_recursive`, `num_lt` contains the number of values in `x` less than `c`, for each candidate, and similarly for `num_eq`.

Then the score is evaluated for each candidate in a loop. The scoring function cannot overflow or return null because the risk of overflow is protected by the preconditions. The function is also completely non-negative due to the use of `abs_diff`.