

fn function_report_noisy_top_k

Michael Shoemate

March 18, 2025

1 Hoare Triple

Precondition

Compiler-verified

Types consistent with pseudocode.

Caller-verified

scale is positive.

Pseudocode

```
1 def function_report_noisy_top_k(  
2     k: int, scale: FBig, optimize: str  
3 ) -> Callable[[list[TIA]], list[int]]:  
4     def function(x: list[TIA]) -> int:  
5         def try_cast(v):  
6             try:  
7                 return FBig.try_from(v)  
8             except Exception:  
9                 return None  
10  
11         # Cast to FBig. #  
12         x = ((i, try_cast(x_i)) for i, x_i in enumerate(x))  
13         # Discard failed casts. #  
14         x = ((i, x_i) for i, x_i in x if x_i is not None)  
15  
16         # Normalize sign. #  
17         y = ((i, -x_i if optimize == "min" else x_i) for i, x_i in x)  
18  
19         # Initialize partial sample. #  
20         def partial_sample(shift):  
21             rv = M0.random_variable(shift, scale) #  
22             return PartialSample.new(rv) #  
23  
24         y = ((i, partial_sample(y_i)) for i, y_i in y)  
25  
26         # Reduce to the k pairs with largest samples. #  
27         def max_sample(a, b):  
28             return a if a[1].greater_than(b[1]) else b  
29  
30         y_top = top(y, k, max_sample) #  
31  
32         # Discard samples, keep indices. #  
33         return [i for i, _ in y_top]  
34
```

Postcondition

Theorem 1.1. Returns a noninteractive function with no side-effects that, when given a vector of non-null scores, returns the indices of the top k z_i , where each $z_i \sim RV(\text{shift} = y_i, \text{scale} = \text{scale})$, and each $y_i = -x_i$ if `optimize` is `min`, else $y_i = x_i$.

The returned function will only return an error if entropy is exhausted. If an error is returned, the error is data-dependent.

Proof. Assume the scores are non-null, as required by the returned function precondition. Therefore casts on line 11 should never fail. However, if the input data is not in the input domain, and a score is null, then line 13 will filter out failed casts. This can be seen as a 1-stable transformation of the input data.

The algorithm then proceeds to line 16. Assuming `optimize` is `max`, the line is a no-op, otherwise it negates each score, therefore each $y_i = -x_i$ if `optimize` is `min`, else $y_i = x_i$.

The algorithm proceeds to line 19. The output measure has an associated noise distribution that is encoded into the Rust type system via `SelectionMeasure`. `MO.random_variable` on line 21 creates a random variable `rv` distributed according to `MO::RV` and parameterized by `shift` (the score), and `scale`.

To sample from this random variable, line 22 constructs an instance of `PartialSample`, which represents an infinitely precise sample from the random variable `rv`. We now have an iterator of pairs containing the index and noisy score of each candidate.

The algorithm proceeds to line 26, which defines a reducer based on `PartialSamplegreater_than`. Assume the scores are non-null, as required by the returned function precondition. Then `max_sample` on line 26 defines a total ordering on the scores.

Since the score vector is finite, and `max_sample` defines a total ordering, then the preconditions for `top` are met. Therefore on line 30 `top` returns the pairs with the top k scores. Line 32 then discards the scores, returning only the indices, which is the desired output.

If entropy is exhausted, then the algorithm will return an error from `PartialSamplegreater_than`. Otherwise, there is one source of error in the function, when there are no non-null scores in the input vector. \square

The algorithm avoids the pitfall of materializing an infinitely precise sample in memory by comparing finite arbitrary-precision bounds on the noisy scores until the lower bound of one noisy score is greater than the upper bound of all others.