`fn gumbel_top_k`

Michael Shoemate

September 14, 2025

1 Hoare Triple

Precondition

Compiler-verified

Types consistent with pseudocode.

- Generic `T` implements `Number`.

Caller-verified

1. Elements of `x` are non-null.
2. `scale` is positive.

Pseudocode

```
def gumbel_top_k(
    x: list[TIA], k: usize, scale: FBig, negate: bool,
) -> list[usize]:
    if scale.is_zero():
        if negate:  #
            cmp = lambda a, b: a < b
        else:
            cmp = lambda a, b: a > b

        def max_sample(a, b):  #
            return a if cmp(a[1], b[1]) else b

        return [i for i, _ in top(x, k, max_sample)]   #

    if all(w[0] == w[1] for w in windows(x, 2)):
        # All values are the same.  #
        x.shuffle()
        return x[:k]

    def try_cast(v):
        try:
            return FBig.try_from(v)
        except Exception:
            return None

    # Cast to FBig.  #
    x = ((i, try_cast(x_i)) for i, x_i in enumerate(x))
    # Discard failed casts.  #
    x = ((i, x_i) for i, x_i in x if x_i is not None)
```

```
30
31      # Normalize sign.  #
32      y = ((i, -x_i if negate else x_i) for i, x_i in x)
33
34      # Initialize partial sample.  #
35      def partial_sample(shift):
36          rv = Gumbel(shift, scale)  #
37          return PartialSample.new(rv)  #
38
39      y = ((i, partial_sample(y_i)) for i, y_i in y)
40
41      # Reduce to the k pairs with largest samples.  #
42      def max_sample(a, b):
43          return a if a[1].greater_than(b[1]) else b
44
45      y_top = top(y, k, max_sample) #
46
47      # Discard samples, keep indices.  #
48      return [i for i, _ in y_top]
```

Postcondition

Theorem 1.1.

- Returns the index of the top element z_i, where each $z_i \sim \text{Gumbel}(\text{shift} = y_i, \text{scale} = \texttt{scale})$, and each $y_i = -x_i$ if `negate`, else $y_i = x_i$, k times with removal.
- Errors are data-independent, except for exhaustion of entropy.

Lemma 1.2. The postcondition holds when the scale is zero.

Proof of Lemma 1.2. The comparator on line 5 flips the sign of scores when `negate` is `true`, therefore each $y_i = -x_i$ if `negate`, else $y_i = x_i$. This avoids negation of a signed integer.

Assume scores are non-null, as required by the precondition. Then `max_sample` on line 10 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 13 `top` returns the pairs with the top k scores. Line 45 then discards the scores, returning only the indices, which is the desired output.

There is one source of error, when there are no non-null scores in the input vector, which is data-independent. □

Lemma 1.3. The postcondition holds when all scores are the same.

Proof of Lemma 1.3. By lemma 1.2, the postcondition holds when the scale is zero.

When all scores are the same, the condition on line 16 is met. The probability of selecting each subset of k candidates is equal, Therefore it is equivalent to randomly select k candidates from the input vector. The algorithm returns the top k after shuffling the input vector, satisfying the postcondition.

The only source of error is due to potential entropy exhaustion. □

Proof of Theorem 1.1. By lemma 1.2, the postcondition holds when the scale is zero, and by lemma 1.3, the postcondition holds when all scores are the same. We now consider the general case.

Assume scores are non-null, as required by the precondition. Therefore casts on line 26 should never fail. However, if the input data is not in the input domain, and a score is null, then line 28 will filter out failed casts. This can be seen as a 1-stable transformation of the input data.

The algorithm then proceeds to line 31. Assuming `negate` is `false`, the line is a no-op, otherwise it negates each score, therefore each $y_i = -x_i$ if `negate` is `true`, else $y_i = x_i$.

The algorithm proceeds to line 34. The output measure has an associated noise distribution that is encoded into the Rust type system via `TopKMeasure`. `MO.random_variable` on line 36 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 37 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 41, which defines a reducer based on `PartialSample`**`greater_than`**. Assume the scores are non-null, as required by the returned function precondition. Then `max_sample` on line 41 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 45 `top` returns the pairs with the top k scores. Line 47 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 `PartialSample`**`greater_than`**. This kind of failure is generally considered data-independent, where a lack of system entropy would occur regardless of the choice of input datasets. However, failure due to lack of entropy can be data-dependent in this case.

An input score vector with all similar scores is expected to require more draws from the random number generator, as the candidates will be very competitive, as compared to a score vector with widely different scores. This technically results in input datasets with more homogeneity being more likely to exhaust entropy and raise an error, violating the data-independent runtime error requirement. This is an unlikely exploit in practice, due to the difficulty of exhausting the RNG's entropy. □

The algorithm avoids 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.