Greenwald-Khanna quantile estimator

November 2, 2021 Mathematics Statistics

The Greenwald-Khanna quantile estimator is a classic sequential quantile estimator which has the following features:

- It allows estimating quantiles with respect to the given precision ϵ.
- It requires $O(\frac{1}{\epsilon}log(\epsilon N))$ memory in the worst case.
- It doesn't require knowledge of the total number of elements in the sequence and the positions of the requested quantiles.

In this post, I briefly explain the basic idea of the underlying data structure, and share a copy-pastable C# implementation. At the end of the post, I discuss some important implementation decisions that are unclear from the original paper, but heavily affect the estimator accuracy.

The main idea of the Greenwald-Khanna quantile estimator

If you want to get a deep understanding of the algorithm I highly recommend reading the original paper [Greenwald2001] first. In this section, we just briefly recall the main idea to provide the context for further implementation.

The quantile estimator has a single parameter ϵ that defines its precision. The suggested data structure supports two operations:

1. `Add(v)`: Add a new element v
2. `GetQuantile(p)`: Get the estimation of the p^{th} quantile

If the number of currently observed elements is n, the max error of `GetQuantile(p)` is ϵn. This means that `GetQuantile` returns an element that belongs to the interval between elements with ranks $(p-\epsilon)n$ and $(p+\epsilon)n$ in the sorted list of observed elements.

The high-level internal representation assumes that we maintain a sorted subset of observed elements: $v_0 \le v_1 \le \ldots \le v_{s-1}$. For each element v_i, we have values $r_{\min}(v_i)$ and $r_{\max}(v_i)$ that denote the minimum and the maximum estimations of the v_i rank across all observed elements. However, maintaining of the actual $r_{\min}(v_i)$ and $r_{\max}(v_i)$ values is computationally inefficient because each `Add` operation would require too many modifications. That's why actually we keep a list of tuples $t_i = (v_i, g_i, \Delta_i)$, where g_i and Δ_i should be interpreted as follows:

$$g_i = r_{\min}(v_i) - r_{\min}(v_{i-1}), \quad \Delta_i = r_{\max}(v_i) - r_{\min}(v_i).$$

The equations could also be described using the following scheme:

```
rmax(v[0]) ←------→ rmax(v[1]) ←------→ rmax(v[2]) ←--...
   ↑                   ↑                   ↑
   | delta[0]          | delta[1]          | delta[2]
   ↓         g[1]      ↓         g[2]      ↓         g[3]
rmin(v[0]) ←------→ rmin(v[1]) ←------→ rmin(v[2]) ←--...
```

Such a representation makes the `Add` operation simple and fast. This operation just inserts a new tuple in a proper place. Since $\{v_i\}$ is sorted, we could find position for the new element using binary search (it requires $O(log(s))$ operations). The value of the corresponding g is always 1. Thus, we "increment" values of $r_{\min}(v)$ and $r_{\max}(v)$ without any actual modifications. The value of the corresponding Δ is $\lfloor 2\epsilon n \rfloor$ for the central elements, and 0 for the border cases (when the new element is the new minimum or the new maximum).

The authors suggest the following invariant for this data structure (assuming we have processed n elements):

$$g_i + \Delta_i \le 2\epsilon n$$

Such an invariant allow us to get a simple implementation of `GetQuantile(p)`. As an estimation of the p^{th} quantile, we could use the element with rank $r = \lceil pn \rceil$. In [Greenwald2001], the authors show that it's always possible to find such a tuple i so that

$$r - \epsilon n \le r_{\min}(v_i), \quad r_{\max}(v_i) \le r + \epsilon n.$$

Since the true rank of v_i across observed elements is inside $[r_{\min}(v_i); r_{\max}(v_i)]$, the difference between the actual rank of v_i and the requested rank r is less than ϵn. Thus, we could use v_i as the estimation of the requested quantile because it satisfies the precision requirements.

The last important aspect of the suggested approach is about maintaining a low number of tuples. Once in a while (after each $\lfloor 1/(2\epsilon) \rfloor$ observed elements), we should perform the `Compress` operation and merge some of the sequential tuple pairs. If we want to merge

tuples t_i and t_{i+1}, the following conditions should be satisfied:

- $g_i + g_{i+1} + \Delta_{i+1} < 2\epsilon n$ (so to maintain the main invariant)
- $\Delta_{i+1} \geq \Delta_i$

If we decide to merge t_i and t_{i+1}, we replace them with a new tuple $t_* = (v_*, g_*, \Delta_*)$ with the following values:

$$v_* = v_{i+1}, \quad g_* = g_i + g_{i+1}, \quad \Delta_* = \Delta_{i+1}.$$

To get additional details about the Greenwald-Khanna quantile estimator, read the original paper [Greenwald2001].

Reference copy-pastable implementation

Based on the suggested description, I came up with the following implementation (it's also available in perfolizer 0.3.0-nightly.107+):

```
public class GreenwaldKhannaQuantileEstimator
{
    private class Tuple
    {
        public static readonly IComparer<Tuple> Comparer =
            Comparer<Tuple>.Create((a, b) => a.Value.CompareTo(b.Value));

        public double Value { get; set; } // Observation v[i]
        public int Gap { get; set; } // g[i] = rMin(v[i]) - rMin(v[i - 1])
        public int Delta { get; set; } // delta[i] = rMax(v[i]) - rMin(v[i])
    }

    private readonly List<Tuple> tuples = new();
    private readonly int compressingInterval;
    private readonly double epsilon;
    private int n;

    public GreenwaldKhannaQuantileEstimator(double epsilon)
    {
        this.epsilon = epsilon;
        compressingInterval = (int)Math.Floor(1.0 / (2.0 * this.epsilon));
    }

    public void Add(double v)
    {
        var t = new Tuple { Value = v, Gap = 1, Delta = (int)Math.Floor(2.0 * epsilon * n) };
        int i = GetInsertIndex(t);
        if (i == 0 || i == tuples.Count)
            t.Delta = 0;

        tuples.Insert(i, t);
        n++;

        if (n % compressingInterval == 0)
            Compress();
    }

    private int GetInsertIndex(Tuple v)
    {
        int index = tuples.BinarySearch(v, Tuple.Comparer);
        return index >= 0 ? index : ~index;
    }

    public double GetQuantile(double p)
    {
        if (tuples.Count == 0)
            throw new InvalidOperationException("Sequence contains no elements");

        double rank = p * (n - 1) + 1;
```

```
        int margin = (int)Math.Ceiling(epsilon * n);

        int bestIndex = -1;
        double bestDist = double.MaxValue;
        int rMin = 0;
        for (int i = 0; i < tuples.Count; i++)
        {
            var t = tuples[i];
            rMin += t.Gap;
            int rMax = rMin + t.Delta;
            if (rank - margin <= rMin && rMax <= rank + margin)
            {
                double currentDist = Math.Abs(rank - (rMin + rMax) / 2.0);
                if (currentDist < bestDist)
                {
                    bestDist = currentDist;
                    bestIndex = i;
                }
            }
        }
        if (bestIndex == -1)
            throw new InvalidOperationException("Failed to find the requested quantile");

        return tuples[bestIndex].Value;
    }

    public void Compress()
    {
        for (int i = tuples.Count - 2; i >= 1; i--)
            while (i < tuples.Count - 1 && DeleteIfNeeded(i))
            {
            }
    }

    private bool DeleteIfNeeded(int i)
    {
        Tuple t1 = tuples[i], t2 = tuples[i + 1];
        int threshold = (int)Math.Floor(2.0 * epsilon * n);
        if (t1.Delta >= t2.Delta && t1.Gap + t2.Gap + t2.Delta < threshold)
        {
            tuples.RemoveAt(i);
            t2.Gap += t1.Gap;
            return true;
        }
        return false;
    }
}
```

Implementation Notes

`GetQuantile(p)`: the expected rank value

The paper suggests (see §2.2.1) that if we want to estimate p^{quantile}, we should find an element with rank $\lceil np \rceil$. Since the paper uses 1-based indexing, we get a problem for $p = 0$. Also, $\lceil np \rceil$ provides a non-uniform mapping from the real quantile position value to the integer rank. I suggest using the same approach that we use in the Hyndman-Fan Type 7 quantile estimator and defining the rank r as a *real* value using the following equation: $r = p(n - 1) + 1$. According to my experiments, it gives more accurate quantile estimations.

`GetQuantile(p)`: the margin value

The paper suggests (see §2.2.1) that we should use ϵn as the search margin. To be more specific, we should find i such that $r - r_{\min}(v_i) \le \epsilon n$ and $r_{\max}(v_i) - r \le \epsilon n$. It's not always possible for small values of ϵ and n (when $\epsilon * n$). I suggest relaxing these conditions and using $\lceil \epsilon n \rceil$ as the margin in the above conditions.

`GetQuantile(p)`: choosing the best tuple

In many cases, several tuples may satisfy the search conditions. However, we should choose a single one to determine the result. If we choose the first one or the last one, the corresponding quantile estimation may be inaccurate (while it technically may satisfy the estimator precision requirements). I suggesting choosing such a tuple t_i so that it minimize the value of $|r - (r_{\min}(v_i) + r_{\max}(v_i))/2|$.

Improving `Compress` routine

The paper suggests (see §2.2.1, Figure 2) the following compress routine:

```
COMPRESS()
  for i from s-2 to 0 do
    if ((BAND(Δ[i], 2εn) ≤ BAND(Δ[i+1], 2εn)) &&
        g[i]+g[i+1]+Δ[i+1] < 2εn) then
          DELETE all descendants of t[i] and the tuple t[i] itself
    end
  end
end
```

I suggest making the following adjustments:

- Replacing the lower bound of the loop by 1. It allows keeping the actual value of the observed minimum so that it provides more accurate estimations of lower quantiles.
- The band representation from §2.1 may help to understand the logic behind the suggested data structure. However, in my opinion, it's not required for the implementation. The condition `(BAND(Δ[i], 2εn) ≤ BAND(Δ[i+1], 2εn)` could be easily replaced by `Δ[i] ≥ Δ[i+1]`.
- The tree representation from §2.1 also looks like an over-complication in the context of implementation. Instead of `DELETE all descendants of t[i] and the tuple t[i] itself` we could just say `MERGE t[i] and t[i+1] while they are mergeable`.

Thus, the above pseudo-code could be rewritten as follows:

```
COMPRESS()
  for i from s-2 to 1 do
    while (i ≤ s-2 &&
           Δ[i] ≥ Δ[i+1] &&
           g[i]+g[i+1]+Δ[i+1] < 2εn) do
             MERGE(t[i], t[i+1])
    end
  end
end
```

References

- **[Greenwald2001]**
  Greenwald, Michael, and Sanjeev Khanna. "Space-efficient online computation of quantile summaries." ACM SIGMOD Record 30, no. 2 (2001): 58-66.
  https://doi.org/10.1145/375663.375670