

COMPUTING EXTREMELY ACCURATE QUANTILES USING t-DIGESTS

TED DUNNING

and otmar ertl

keywords

quantile, t-digest, ted, centroid, monoid, dunning, percentile, ted, really, crdt, math, median, quantile, ted

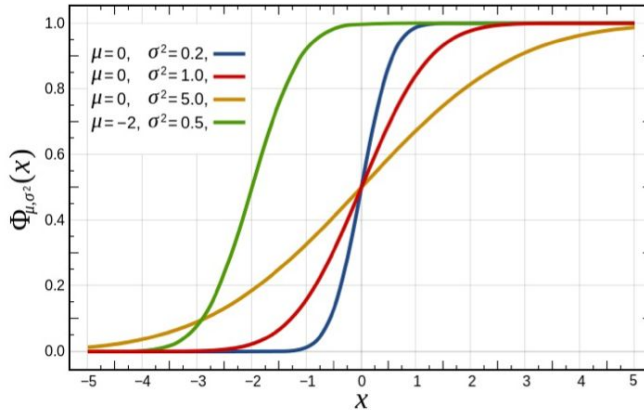
TED

IDEAS WORTH SPREADING

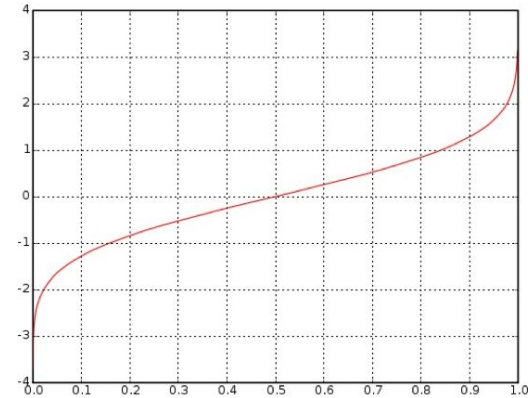


Cumulative Distribution Function and Quantile Function

- CDF: what portion of samples are less than a given value
- Quantile: what value is greater than a given portion of samples
- Quantile is inverse of CDF



CDF of Gaussian



Quantile of Gaussian

CDF Table

SAT Percentile Ranks for Males, Females, and Total Group

2006 College-Bound Seniors—Critical Reading + Mathematics + Writing

Score	Total		Male		Female	
	Number	Percentile	Number	Percentile	Number	Percentile
2400	238	99+	131	99+	107	99+
2390	240	99+	122	99+	118	99+
2380	153	99+	81	99+	72	99+
2370	227	99+	122	99+	105	99+
2360	402	99+	213	99+	189	99+
2350	330	99+	197	99+	133	99+
2340	534	99+	288	99+	246	99+
2330	532	99+	295	99+	237	99+
2320	579	99+	319	99+	260	99+
2310	758	99+	405	99+	353	99+
2300	872	99+	446	99+	426	99+
2290	848	99+	481	99+	367	99+
2280	1,019	99+	528	99	491	99+
2270	1,048	99	577	99	471	99+
2260	1,209	99	620	99	589	99
2250	1,394	99	737	99	657	99
2240	1,428	99	742	99	686	99
2230	1,572	99	821	99	751	99
2220	1,729	99	914	99	815	99
2210	1,850	99	961	99	889	99
2200	2,022	99	1,053	98	969	99
2190	1,992	98	996	98	996	99
2180	2,214	98	1,169	98	1,045	99
2170	2,342	98	1,182	98	1,160	98
2160	2,506	98	1,298	98	1,208	98
2150	2,622	98	1,366	97	1,256	98
2140	2,939	98	1,519	97	1,420	98
2130	3,058	97	1,570	97	1,488	98
2120	3,190	97	1,640	97	1,550	97
2110	3,432	97	1,798	96	1,634	97
2100	3,698	97	1,894	96	1,804	97
2090	3,796	96	1,942	96	1,854	97
2080	3,953	96	1,987	96	1,966	96
2070	4,195	96	2,116	95	2,079	96
2060	4,315	95	2,232	95	2,083	96
2050	4,540	95	2,374	94	2,166	96

Quantile Table

SAT/ACT

SCORE COMPARISON CHART

	SAT	ACT
99%	2200+	32+
98%	2140	31
97%	2100	30
95%	2040	29
89%	1910	27
85%	1850	26
81%	1800	25
75%	1730	24
69%	1670	23
63%	1610	22
56%	1550	21
48%	1490	20
40%	1430	19
33%	1370	18
26%	1310	17
19%	1240	16
13%	1170	15
08%	1090	14
05%	1010	13
02%	880	12
01%	870	11

Application: Service Latency

- Running mean, variance are easy to calculate, but not useful
- Latency quantiles better categorize overall user experience
- An exact quantile calculation requires a sorted list of all samples
- At scale, collecting and storing these samples is expensive



4

1

0

3

2

5

4

9

7

3

4

1

0

0

0

0

0

0

0

0

4

1

1

1

1

1

1

1

1

4

3

2.5^2

2.5^2

2.5^2

2.5^2

2.5^2

$2.\overline{6}^3$

4

4

4

4^2

4^2

4^2

4^2

5

5

5

5

5

9

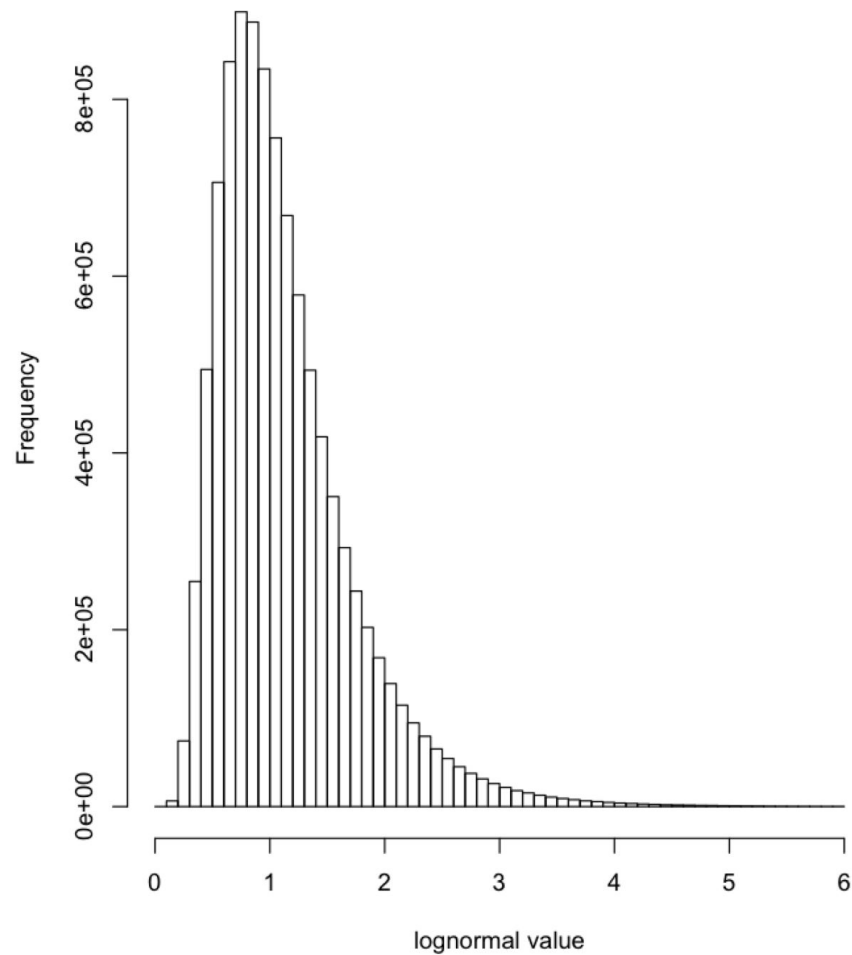
7

7

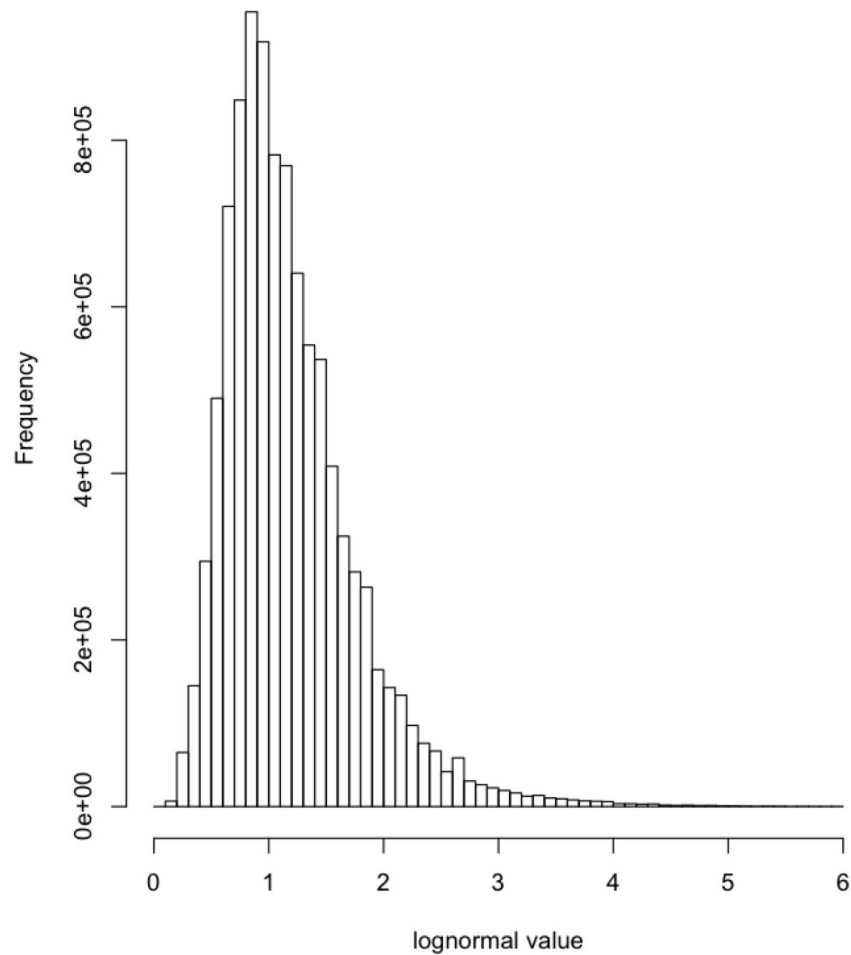
9

9

10M samples



12KB t-digest














t-Digest as Almost-Monoid

- For efficient insertion, t-Digest must actually estimate quantile (q)
- $q(x)$ can be calculated by summing up weights of centroids $< x$
- These partial sums are associative, so monoid
- A monoidal tree could be used to store ordered centroids
- Catch: centroid values change during insertion (merge)
- Fortunately, changes do not affect the relative position of the centroid

 **tdunning** committed on **GitHub** Merge pull request [#66](#) from poojke/master ...

Latest commit [a4f30ff](#) on Jun 17

..

 AVLGroupTree.java	add AVLGroupTreei default constructor to support Kryo serialization u...	2 months ago
 AVLTreeDigest.java	Simplify loop introduced in #44	a year ago
 AbstractTDigest.java	Closes #32. Switch to randomized tests (thanks to jpountz)	2 months ago
 ArrayDigest.java	Make various TDigest implementations Serializable (so that it can be ...	a year ago
 Centroid.java	Closes #68: Make centroid ID's volatile to avoid confusion when movin...	2 months ago
 GroupTree.java	Make various TDigest implementations Serializable (so that it can be ...	a year ago
 IntAVLTree.java	Make various TDigest implementations Serializable (so that it can be ...	a year ago
 MergingDigest.java	Switch to virtual collection for centroids()	a year ago
 Sort.java	More cleanups for the MergingDigest. Not quite there yet.	a year ago
 TDigest.java	Merge pull request #56 from rayortigas/serializable	2 months ago
 TreeDigest.java	Closes #32. Switch to randomized tests (thanks to jpountz)	2 months ago

t-Digest CRDT (Commutative Replicated Dunning, Ted)

- t-Digests are great, but how to merge them from multiple sources?
- Centroid exchange must be Commutative, Associative, and Idempotent
- Merge centroids by inserting from both peers into an empty t-Digest
- Halve counts from both for numeric stability (count → weight)
- Need a new way to represent total sample counts: G-Counter

Example: peers P1, P2

digests D1, D2: [{:value float, :weight float}...]

counters C1, C2: {:P1 integer, :P2 integer, ...}

Before:

P1: D1: [{:value 1, :weight 2}, {:value 3, :weight 4}]

C1: {P1 6}

P2: D2: [{:value 0, :weight 1}, {:value 2, :weight 6}]

C2: {P2 7}

After:

D1 = D2: [{:value 0, :weight 0.5},
{:value 1.5, :weight 4},
{:value 3, :weight 2}]

C1 = C2: {P1 6, P2 7}

t-Digest Rotation

- Computing running quantiles over all of time isn't very useful
- SLAs can be specified as maximum of quantile within an interval
- Ex: 99% of requests per hour are served within 500ms
- Need a way to roll over results but still report after rollover

Solution:

- Maintain a pair of t-Digest data structures: **old** and **new**
- Write all incoming samples to both t-Digests
- At rollover time, **old** = **new**, **new** = empty
- Always output results using **old**

Example: 1 hour window (30 minute rollover)

- At 6:59, **old** has samples from 6:00–6:59, **new** has 6:30–6:59
- Rollover at 7:00, **old** has 6:30–7:00, **new** has 7:00–
- At 7:29, **old** has 6:30–7:29, **new** has 7:00–7:29

```

51  /**
52  * Standard quick sort except that sorting is done on an index array rather than the values themselves
53  *
54  * @param order The pre-allocated index array
55  * @param values The values to sort
56  * @param start The beginning of the values to sort
57  * @param end The value after the last value to sort
58  * @param limit The minimum size to recurse down to.
59  */
60  private static void quickSort(int[] order, double[] values, int start, int end, int limit) {
61      // the while loop implements tail-recursion to avoid excessive stack calls on nasty cases
62      while (end - start > limit) {
63
64          // median of three values for the pivot
65          int a = start;
66          int b = (start + end) / 2;
67          int c = end - 1;
68
69          int pivotIndex;
70          double pivotValue;
71          double va = values[order[a]];
72          double vb = values[order[b]];
73          double vc = values[order[c]];
74          if (va > vb) {
75              if (vc > va) {
76                  // vc > va > vb
77                  pivotIndex = a;
78                  pivotValue = va;
79              } else {
80                  // va > vb, va > vc
81                  if (vc < vb) {
82                      // va > vb > vc
83                      pivotIndex = b;
84                      pivotValue = vb;
85                  } else {
86                      // va >= va, vb >= vc
87                      pivotIndex = c;
88                      pivotValue = vc;
89                  }
90              }
91          } else {
92              // vb >= va
93              if (vc > vb) {
94                  // vc > vb >= va
95                  pivotIndex = b;
96                  pivotValue = vb;
97              } else {
98                  // vb >= va, vb >= vc
99                  if (vc < va) {
100                      // vb >= va > vc
101                      pivotIndex = a;
102                      pivotValue = va;
103                  } else {
104                      // vb >= vc >= va
105                      pivotIndex = c;
106                      pivotValue = vc;
107                  }
108              }
109          }
110
111          // move pivot to beginning of array
112          swap(order, start, pivotIndex);
113
114          // we use a three way partition because many duplicate values is an important case
115

```

```

116  int low = start + 1; // low points to first value not known to be equal to pivotValue
117  int high = end; // high points to first value > pivotValue
118  int i = low; // i scans the array
119  while (i < high) {
120      // invariant: values[order[k]] == pivotValue for k in [0..low)
121      // invariant: values[order[k]] < pivotValue for k in [low..i)
122      // invariant: values[order[k]] > pivotValue for k in [high..end)
123      // in-loop: i < high
124      // in-loop: low < high
125      // in-loop: i >= low
126      double vi = values[order[i]];
127      if (vi == pivotValue) {
128          if (low != i) {
129              swap(order, low, i);
130          } else {
131              i++;
132          }
133          low++;
134      } else if (vi > pivotValue) {
135          high--;
136          swap(order, i, high);
137      } else {
138          // vi < pivotValue
139          i++;
140      }
141  }
142  // invariant: values[order[k]] == pivotValue for k in [0..low)
143  // invariant: values[order[k]] < pivotValue for k in [low..i)
144  // invariant: values[order[k]] > pivotValue for k in [high..end)
145  // assert i == high || low == high therefore, we are done with partition
146
147  // at this point, i==high, from [start,low) are == pivot, [low,high) are < and [high,end) are >
148  // we have to move the values equal to the pivot into the middle. To do this, we swap pivot
149  // values into the top end of the [low,high) range stopping when we run out of destinations
150  // or when we run out of values to copy
151  int from = start;
152  int to = high - 1;
153  for (i = 0; from < low && to >= low; i++) {
154      swap(order, from++, to--);
155  }
156  if (from == low) {
157      // ran out of things to copy. This means that the last destination is the boundary
158      low = to + 1;
159  } else {
160      // ran out of places to copy to. This means that there are uncopied pivots and the
161      // boundary is at the beginning of those
162      low = from;
163  }
164
165  // checkPartition(order, values, pivotValue, start, low, high, end);
166
167  // now recurse, but arrange it so we handle the longer limit by tail recursion
168  if (low - start < end - high) {
169      quickSort(order, values, start, low, limit);
170
171      // this is really a way to do
172      // quickSort(order, values, high, end, limit);
173      start = high;
174  } else {
175      quickSort(order, values, high, end, limit);
176      // this is really a way to do
177      // quickSort(order, values, start, low, limit);
178      end = low;
179  }
180  }
181  }

```

