package com.twitter.search.common.encoding.features;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

/\*\*

\* A smart integer normalizer that converts an integer of a known range to a small integer up to

\* 8 bits long. This normalizer generates a boundary value array in the constructor as the buckets

\* for different values.

\* <p/>

\* The normalized value has a nice properties:

\* 1) it maintains the order of original value: if a > b, then normalize(a) > normalize(b).

\* 2) the value 0 is always normalized to byte 0.

\* 3) the normalized values are (almost) evenly distributed on the log scale

\* 4) no waste in code space, all possible values representable by normalized bits are used,

\* each corresponding to a different value.

\*/

public class SmartIntegerNormalizer extends ByteNormalizer {

// The max value we want to support in this normalizer. If the input is larger than this value,

// it's normalized as if it's the maxValue.

private final int maxValue;

// Number of bits used for normalized value, the largest normalized value

// would be (1 << numBits) - 1.

private final int numBits;

// The inclusive lower bounds of all buckets. A normalized value k corresponds to original values

// in the inclusive-exclusive range

// [ boundaryValues[k], boundaryValues[k+1] )

private final int[] boundaryValues;

// The length of the boundaryValues array, or the number of buckets.

private final int length;

/\*\*

\* Construct a normalizer.

\*

\* @param maxValue max value it supports, must be larger than minValue. Anything larger than this

\* would be treated as maxValue.

\* @param numBits number of bits you want to use for this normalization, between 1 and 8.

\* higher resolution for the lower numbers.

\*/

public SmartIntegerNormalizer(int maxValue, int numBits) {

Preconditions.checkArgument(maxValue > 0);

Preconditions.checkArgument(numBits > 0 && numBits <= 8);

this.maxValue = maxValue;

this.numBits = numBits;

this.length = 1 << numBits;

this.boundaryValues = new int[length];

int index;

for (index = length - 1; index >= 0; --index) {

// values are evenly distributed on the log scale

int boundary = (int) Math.pow(maxValue, (double) index / length);

// we have more byte slots left than we have possible boundary values (buckets),

// just give consecutive boundary values to all remaining slots, starting from 0.

if (boundary <= index) {

break;

}

boundaryValues[index] = boundary;

}

if (index >= 0) {

for (int i = 1; i <= index; ++i) {

boundaryValues[i] = i;

}

}

boundaryValues[0] = 0; // the first one is always 0.

}

@Override

public byte normalize(double val) {

int intVal = (int) (val > maxValue ? maxValue : val);

return intToUnsignedByte(binarySearch(intVal, boundaryValues));

}

/\*\*

\* Return the lower bound of the bucket represent by norm. This simply returns the boundary

\* value indexed by current norm.

\*/

@Override

public double unnormLowerBound(byte norm) {

return boundaryValues[unsignedByteToInt(norm)];

}

/\*\*

\* Return the upper bound of the bucket represent by norm. This returns the next boundary value

\* minus 1. If norm represents the last bucket, it returns the maxValue.

\*/

@Override

public double unnormUpperBound(byte norm) {

// if it's already the last possible normalized value, just return the corresponding last

// boundary value.

int intNorm = unsignedByteToInt(norm);

if (intNorm == length - 1) {

return maxValue;

}

return boundaryValues[intNorm + 1] - 1;

}

/\*\*

\* Do a binary search on array and find the index of the item that's no bigger than value.

\*/

private static int binarySearch(int value, int[] array) {

// corner cases

if (value <= array[0]) {

return 0;

} else if (value >= array[array.length - 1]) {

return array.length - 1;

}

int left = 0;

int right = array.length - 1;

int pivot = (left + right) >> 1;

do {

int midVal = array[pivot];

if (value == midVal) {

break;

} else if (value > midVal) {

left = pivot;

} else {

right = pivot;

}

pivot = (left + right) >> 1;

} while (pivot != left);

return pivot;

}

@Override

public String toString() {

StringBuilder sb = new StringBuilder(String.format(

"Smart Integer Normalizer (numBits = %d, max = %d)\n",

this.numBits, this.maxValue));

for (int i = 0; i < this.length; i++) {

sb.append(String.format(

"[%2d] boundary = %6d, range [ %6d, %6d ), norm: %4d | %4d | %4d %s\n",

i, boundaryValues[i],

(int) unnormLowerBound(intToUnsignedByte(i)),

(int) unnormUpperBound(intToUnsignedByte(i)),

unsignedByteToInt(normalize(boundaryValues[i] - 1)),

unsignedByteToInt(normalize(boundaryValues[i])),

unsignedByteToInt(normalize(boundaryValues[i] + 1)),

i == boundaryValues[i] ? "\*" : ""));

}

return sb.toString();

}

@VisibleForTesting

int[] getBoundaryValues() {

return boundaryValues;

}

}