package com.twitter.frigate.pushservice.predicate.ntab\_caret\_fatigue

import com.twitter.finagle.stats.StatsReceiver

case class ContinuousFunctionParam(

knobs: Seq[Double],

knobValues: Seq[Double],

powers: Seq[Double],

weight: Double,

defaultValue: Double) {

def validateParams(): Boolean = {

knobs.size > 0 && knobs.size - 1 == powers.size && knobs.size == knobValues.size

}

}

object ContinuousFunction {

/\*\*

\* Evalutate the value for function f(x) = w(x - b)^power

\* where w and b are decided by the start, startVal, end, endVal

\* such that

\* w(start - b) ^ power = startVal

\* w(end - b) ^ power = endVal

\*

\* @param value the value at which we will evaluate the param

\* @return weight \* f(value)

\*/

def evaluateFn(

value: Double,

start: Double,

startVal: Double,

end: Double,

endVal: Double,

power: Double,

weight: Double

): Double = {

val b =

(math.pow(startVal / endVal, 1 / power) \* end - start) / (math.pow(

startVal / endVal,

1 / power) - 1)

val w = startVal / math.pow(start - b, power)

weight \* w \* math.pow(value - b, power)

}

/\*\*

\* Evaluate value for function f(x), and return weight \* f(x)

\*

\* f(x) is a piecewise function

\* f(x) = w\_i \* (x - b\_i)^powers[i] for knobs[i] <= x < knobs[i+1]

\* such that

\* w(knobs[i] - b) ^ power = knobVals[i]

\* w(knobs[i+1] - b) ^ power = knobVals[i+1]

\*

\* @return Evaluate value for weight \* f(x), for the function described above. If the any of the input is invalid, returns defaultVal

\*/

def safeEvaluateFn(

value: Double,

knobs: Seq[Double],

knobVals: Seq[Double],

powers: Seq[Double],

weight: Double,

defaultVal: Double,

statsReceiver: StatsReceiver

): Double = {

val totalStats = statsReceiver.counter("safe\_evalfn\_total")

val validStats =

statsReceiver.counter("safe\_evalfn\_valid")

val validEndCaseStats =

statsReceiver.counter("safe\_evalfn\_valid\_endcase")

val invalidStats = statsReceiver.counter("safe\_evalfn\_invalid")

totalStats.incr()

if (knobs.size <= 0 || knobs.size - 1 != powers.size || knobs.size != knobVals.size) {

invalidStats.incr()

defaultVal

} else {

val endIndex = knobs.indexWhere(knob => knob > value)

validStats.incr()

endIndex match {

case -1 => {

validEndCaseStats.incr()

knobVals(knobVals.size - 1) \* weight

}

case 0 => {

validEndCaseStats.incr()

knobVals(0) \* weight

}

case \_ => {

val startIndex = endIndex - 1

evaluateFn(

value,

knobs(startIndex),

knobVals(startIndex),

knobs(endIndex),

knobVals(endIndex),

powers(startIndex),

weight)

}

}

}

}

def safeEvaluateFn(

value: Double,

fnParams: ContinuousFunctionParam,

statsReceiver: StatsReceiver

): Double = {

val totalStats = statsReceiver.counter("safe\_evalfn\_total")

val validStats =

statsReceiver.counter("safe\_evalfn\_valid")

val validEndCaseStats =

statsReceiver.counter("safe\_evalfn\_valid\_endcase")

val invalidStats = statsReceiver.counter("safe\_evalfn\_invalid")

totalStats.incr()

if (fnParams.validateParams()) {

val endIndex = fnParams.knobs.indexWhere(knob => knob > value)

validStats.incr()

endIndex match {

case -1 => {

validEndCaseStats.incr()

fnParams.knobValues(fnParams.knobValues.size - 1) \* fnParams.weight

}

case 0 => {

validEndCaseStats.incr()

fnParams.knobValues(0) \* fnParams.weight

}

case \_ => {

val startIndex = endIndex - 1

evaluateFn(

value,

fnParams.knobs(startIndex),

fnParams.knobValues(startIndex),

fnParams.knobs(endIndex),

fnParams.knobValues(endIndex),

fnParams.powers(startIndex),

fnParams.weight

)

}

}

} else {

invalidStats.incr()

fnParams.defaultValue

}

}

}