package com.twitter.follow\_recommendations.common.utils

import scala.util.Random

object RandomUtil {

/\*\*

\* Takes a seq of items which have weights. Returns an infinite stream that is

\* sampled with replacement using the weights for each item. All weights need

\* to be greater than or equal to zero. In addition, the sum of weights

\* should be greater than zero.

\*

\* @param items items

\* @param weighted provides weights for items

\* @tparam T type of item

\* @return Stream of Ts

\*/

def weightedRandomSamplingWithReplacement[T](

items: Seq[T],

random: Option[Random] = None

)(

implicit weighted: Weighted[T]

): Stream[T] = {

if (items.isEmpty) {

Stream.empty

} else {

val weights = items.map { i => weighted(i) }

assert(weights.forall { \_ >= 0 }, "Negative weight exists for sampling")

val cumulativeWeight = weights.scanLeft(0.0)(\_ + \_).tail

assert(cumulativeWeight.last > 0, "Sum of the sampling weights is not positive")

val cumulativeProbability = cumulativeWeight map (\_ / cumulativeWeight.last)

def next(): Stream[T] = {

val rand = random.getOrElse(Random).nextDouble()

val idx = cumulativeProbability.indexWhere(\_ >= rand)

items(if (idx == -1) items.length - 1 else idx) #:: next()

}

next()

}

}

/\*\*

\* Takes a seq of items and their weights. Returns a lazy weighted shuffle of

\* the elements in the list. All weights should be greater than zero.

\*

\* @param items items

\* @param weighted provides weights for items

\* @tparam T type of item

\* @return Stream of Ts

\*/

def weightedRandomShuffle[T](

items: Seq[T],

random: Option[Random] = None

)(

implicit weighted: Weighted[T]

): Stream[T] = {

assert(items.forall { i => weighted(i) > 0 }, "Non-positive weight exists for shuffling")

def next(it: Seq[T]): Stream[T] = {

if (it.isEmpty)

Stream.empty

else {

val cumulativeWeight = it.scanLeft(0.0)((acc: Double, curr: T) => acc + weighted(curr)).tail

val cutoff = random.getOrElse(Random).nextDouble() \* cumulativeWeight.last

val idx = cumulativeWeight.indexWhere(\_ >= cutoff)

val (left, right) = it.splitAt(idx)

it(if (idx == -1) it.size - 1 else idx) #:: next(left ++ right.drop(1))

}

}

next(items)

}

/\*\*

\* Takes a seq of items and a weight function, returns a lazy weighted shuffle of

\* the elements in the list.The weight function is based on the rank of the element

\* in the original lst.

\* @param items

\* @param rankToWeight

\* @param random

\* @tparam T

\* @return

\*/

def weightedRandomShuffleByRank[T](

items: Seq[T],

rankToWeight: Int => Double,

random: Option[Random] = None

): Stream[T] = {

val candWeights = items.zipWithIndex.map { case (item, rank) => (item, rankToWeight(rank)) }

RandomUtil.weightedRandomShuffle(candWeights, random).map(\_.\_1)

}

}