package com.twitter.follow\_recommendations.common.transforms.weighted\_sampling

import com.twitter.follow\_recommendations.common.base.GatedTransform

import com.twitter.stitch.Stitch

import com.twitter.timelines.configapi.HasParams

import com.twitter.follow\_recommendations.common.models.CandidateUser

import com.twitter.follow\_recommendations.common.models.HasDebugOptions

import com.twitter.follow\_recommendations.common.models.Score

import com.twitter.follow\_recommendations.common.models.Scores

import com.twitter.follow\_recommendations.common.rankers.common.RankerId

import com.twitter.follow\_recommendations.common.rankers.utils.Utils

import com.twitter.product\_mixer.core.model.marshalling.request.HasClientContext

import javax.inject.Inject

import javax.inject.Singleton

@Singleton

class SamplingTransform @Inject() ()

extends GatedTransform[HasClientContext with HasParams with HasDebugOptions, CandidateUser] {

val name: String = this.getClass.getSimpleName

/\*

Description: This function takes in a set of candidate users and ranks them for a who-to-follow

request by sampling from the Placket-Luce distribution

(https://cran.rstudio.com/web/packages/PlackettLuce/vignettes/Overview.html) with a three

variations. The first variation is that the scores of the candidates are multiplied by

multiplicativeFactor before sampling. The second variation is that the scores are

exponentiated before sampling. The third variation is that depending on how many who-to-follow

positions are being requested, the first k positions are reserved for the candidates with the

highest scores (and they are sorted in decreasing order of score) and the remaining positions

are sampled from a Placket-Luce. We use the efficient algorithm proposed in this blog

https://medium.com/swlh/going-old-school-designing-algorithms-for-fast-weighted-sampling-in-production-c48fc1f40051

to sample from a Plackett-Luce. Because of numerical stability reasons, before sampling from this

distribution, (1) we subtract off the maximum score from all the scores and (2) if after

this subtraction and multiplication by the multiplicative factor the resulting score is <= -10,

we force the candidate's transformed score under the above algorithm to be 0 (so r^(1/w) = 0)

where r is a random number and w is the transformed score.

inputs:

- target: HasClientContext (WTF request)

- candidates: sequence of CandidateUsers (users that need to be ranked from a who-to-follow

request) each of which has a score

inputs accessed through feature switches, i.e. through target.params (see the following file:

"follow-recommendations-service/common/src/main/scala/com/twitter/follow\_recommendations/common/

transforms/weighted\_sampling/SamplingTransformParams.scala"):

- topKFixed: the first k positions of the who-to-follow ranking correspond to the users with the k

highest scores and are not sampled from the Placket-Luce distribution

- multiplicativeFactor: multiplicativeFactor is used to transform the scores of each candidate by

multiplying that user's score by multiplicativeFactor

output:

- Sequence of CandidateUser whose order represents the ranking of users in a who-to-follow request

This ranking is sampled from a Placket-Luce distribution.

\*/

override def transform(

target: HasClientContext with HasParams with HasDebugOptions,

candidates: Seq[CandidateUser]

): Stitch[Seq[CandidateUser]] = {

// the first k positions of the who-to-follow ranking correspond to the users with the k

// highest scores and are not sampled from the Placket-Luce distribution

val topKFixed = target.params(SamplingTransformParams.TopKFixed)

// multiplicativeFactor is used to transform the scores of each candidate by

// multiplying that user's score by multiplicativeFactor

val multiplicativeFactor = target.params(SamplingTransformParams.MultiplicativeFactor)

// sort candidates by their score

val candidatesSorted = candidates.sortBy(-1 \* \_.score.getOrElse(0.0))

// pick the top K candidates by score and the remaining candidates

val (topKFixedCandidates, candidatesOutsideOfTopK) =

candidatesSorted.zipWithIndex.partition { case (value, index) => index < topKFixed }

val randomNumGenerator =

new scala.util.Random(target.getRandomizationSeed.getOrElse(System.currentTimeMillis))

// we need to subtract the maximum score off the scores for numerical stability reasons

// subtracting the max score off does not effect the underlying distribution we are sampling

// the candidates from

// we need the if statement since you cannot take the max of an empty sequence

val maximum\_score = if (candidatesOutsideOfTopK.nonEmpty) {

candidatesOutsideOfTopK.map(x => x.\_1.score.getOrElse(0.0)).max

} else {

0.0

}

// for candidates in candidatesOutsideOfTopK, we transform their score by subtracting off

// maximum\_score and then multiply by multiplicativeFactor

val candidatesOutsideOfTopKTransformedScore = candidatesOutsideOfTopK.map(x =>

(x.\_1, multiplicativeFactor \* (x.\_1.score.getOrElse(0.0) - maximum\_score)))

// for each candidate with score transformed and clip score w, sample a random number r,

// create a new score r^(1/w) and sort the candidates to get the final ranking.

// for numerical stability reasons if the score is <=-10, we force r^(1/w) = 0.

// this samples the candidates from the modified Plackett-Luce distribution. See

// https://medium.com/swlh/going-old-school-designing-algorithms-for-fast-weighted-sampling-in-production-c48fc1f40051

val candidatesOutsideOfTopKSampled = candidatesOutsideOfTopKTransformedScore

.map(x =>

(

x.\_1,

if (x.\_2 <= -10.0)

0.0

else

scala.math.pow(

randomNumGenerator.nextFloat(),

1 / (scala.math

.exp(x.\_2))))).sortBy(-1 \* \_.\_2)

val topKCandidates: Seq[CandidateUser] = topKFixedCandidates.map(\_.\_1)

val scribeRankingInfo: Boolean =

target.params(SamplingTransformParams.ScribeRankingInfoInSamplingTransform)

val transformedCandidates: Seq[CandidateUser] = if (scribeRankingInfo) {

val topKCandidatesWithRankingInfo: Seq[CandidateUser] =

Utils.addRankingInfo(topKCandidates, name)

val candidatesOutsideOfTopKSampledWithRankingInfo: Seq[CandidateUser] =

candidatesOutsideOfTopKSampled.zipWithIndex.map {

case ((candidate, score), rank) =>

val newScore = Seq(Score(score, Some(RankerId.PlacketLuceSamplingTransformer)))

val newScores: Option[Scores] = candidate.scores

.map { scores =>

scores.copy(scores = scores.scores ++ newScore)

}.orElse(Some(Scores(newScore, Some(RankerId.PlacketLuceSamplingTransformer))))

val globalRank = rank + topKFixed + 1

candidate.addInfoPerRankingStage(name, newScores, globalRank)

}

topKCandidatesWithRankingInfo ++ candidatesOutsideOfTopKSampledWithRankingInfo

} else {

topKCandidates ++ candidatesOutsideOfTopKSampled.map(\_.\_1)

}

Stitch.value(transformedCandidates)

}

}