package com.twitter.timelines.data\_processing.ad\_hoc.earlybird\_ranking.model\_evaluation

import scala.collection.GenTraversableOnce

case class CandidateRecord(tweetId: Long, fullScore: Double, earlyScore: Double, served: Boolean)

/\*\*

\* A metric that compares scores generated by a "full" prediction

\* model to a "light" (Earlybird) model. The metric is calculated for candidates

\* from a single request.

\*/

sealed trait EarlybirdEvaluationMetric {

def name: String

def apply(candidates: Seq[CandidateRecord]): Option[Double]

}

/\*\*

\* Picks the set of `k` top candidates using light scores, and calculates

\* recall of these light-score based candidates among set of `k` top candidates

\* using full scores.

\*

\* If there are fewer than `k` candidates, then we can choose to filter out requests (will

\* lower value of recall) or keep them by trivially computing recall as 1.0.

\*/

case class TopKRecall(k: Int, filterFewerThanK: Boolean) extends EarlybirdEvaluationMetric {

override val name: String = s"top\_${k}\_recall${if (filterFewerThanK) "\_filtered" else ""}"

override def apply(candidates: Seq[CandidateRecord]): Option[Double] = {

if (candidates.size <= k) {

if (filterFewerThanK) None else Some(1.0)

} else {

val topFull = candidates.sortBy(-\_.fullScore).take(k)

val topLight = candidates.sortBy(-\_.earlyScore).take(k)

val overlap = topFull.map(\_.tweetId).intersect(topLight.map(\_.tweetId))

val truePos = overlap.size.toDouble

Some(truePos / k.toDouble)

}

}

}

/\*\*

\* Calculates the probability that a random pair of candidates will be ordered the same by the

\* full and earlybird models.

\*

\* Note: A pair with same scores for one model and different for the other will contribute 1

\* to the sum. Pairs that are strictly ordered the same, will contribute 2.

\* It follows that the score for a constant model is 0.5, which is approximately equal to a

\* random model as expected.

\*/

case object ProbabilityOfCorrectOrdering extends EarlybirdEvaluationMetric {

def fractionOf[A](trav: GenTraversableOnce[A])(p: A => Boolean): Double = {

if (trav.isEmpty)

0.0

else {

val (numPos, numElements) = trav.foldLeft((0, 0)) {

case ((numPosAcc, numElementsAcc), elem) =>

(if (p(elem)) numPosAcc + 1 else numPosAcc, numElementsAcc + 1)

}

numPos.toDouble / numElements

}

}

override def name: String = "probability\_of\_correct\_ordering"

override def apply(candidates: Seq[CandidateRecord]): Option[Double] = {

if (candidates.size < 2)

None

else {

val pairs = for {

left <- candidates.iterator

right <- candidates.iterator

if left != right

} yield (left, right)

val probabilityOfCorrect = fractionOf(pairs) {

case (left, right) =>

(left.fullScore > right.fullScore) == (left.earlyScore > right.earlyScore)

}

Some(probabilityOfCorrect)

}

}

}

/\*\*

\* Like `TopKRecall`, but uses `n` % of top candidates instead.

\*/

case class TopNPercentRecall(percent: Double) extends EarlybirdEvaluationMetric {

override val name: String = s"top\_${percent}\_pct\_recall"

override def apply(candidates: Seq[CandidateRecord]): Option[Double] = {

val k = Math.floor(candidates.size \* percent).toInt

if (k > 0) {

val topFull = candidates.sortBy(-\_.fullScore).take(k)

val topLight = candidates.sortBy(-\_.earlyScore).take(k)

val overlap = topFull.map(\_.tweetId).intersect(topLight.map(\_.tweetId))

val truePos = overlap.size.toDouble

Some(truePos / k.toDouble)

} else {

None

}

}

}

/\*\*

\* Picks the set of `k` top candidates using light scores, and calculates

\* recall of selected light-score based candidates among set of actual

\* shown candidates.

\*/

case class ShownTweetRecall(k: Int) extends EarlybirdEvaluationMetric {

override val name: String = s"shown\_tweet\_recall\_$k"

override def apply(candidates: Seq[CandidateRecord]): Option[Double] = {

if (candidates.size <= k) {

None

} else {

val topLight = candidates.sortBy(-\_.earlyScore).take(k)

val truePos = topLight.count(\_.served).toDouble

val allPos = candidates.count(\_.served).toDouble

if (allPos > 0) Some(truePos / allPos)

else None

}

}

}

/\*\*

\* Like `ShownTweetRecall`, but uses `n` % of top candidates instead.

\*/

case class ShownTweetPercentRecall(percent: Double) extends EarlybirdEvaluationMetric {

override val name: String = s"shown\_tweet\_recall\_${percent}\_pct"

override def apply(candidates: Seq[CandidateRecord]): Option[Double] = {

val k = Math.floor(candidates.size \* percent).toInt

val topLight = candidates.sortBy(-\_.earlyScore).take(k)

val truePos = topLight.count(\_.served).toDouble

val allPos = candidates.count(\_.served).toDouble

if (allPos > 0) Some(truePos / allPos)

else None

}

}

/\*\*

\* Like `ShownTweetRecall`, but calculated using \*full\* scores. This is a sanity metric,

\* because by definition the top full-scored candidates will be served. If the value is

\* < 1, this is due to the ranked section being smaller than k.

\*/

case class ShownTweetRecallWithFullScores(k: Int) extends EarlybirdEvaluationMetric {

override val name: String = s"shown\_tweet\_recall\_with\_full\_scores\_$k"

override def apply(candidates: Seq[CandidateRecord]): Option[Double] = {

if (candidates.size <= k) {

None

} else {

val topFull = candidates.sortBy(-\_.fullScore).take(k)

val truePos = topFull.count(\_.served).toDouble

val allPos = candidates.count(\_.served).toDouble

if (allPos > 0) Some(truePos / allPos)

else None

}

}

}

/\*\*

\* Picks the set of `k` top candidates using the light scores, and calculates

\* average full score for the candidates.

\*/

case class AverageFullScoreForTopLight(k: Int) extends EarlybirdEvaluationMetric {

override val name: String = s"average\_full\_score\_for\_top\_light\_$k"

override def apply(candidates: Seq[CandidateRecord]): Option[Double] = {

if (candidates.size <= k) {

None

} else {

val topLight = candidates.sortBy(-\_.earlyScore).take(k)

Some(topLight.map(\_.fullScore).sum / topLight.size)

}

}

}

/\*\*

\* Picks the set of `k` top candidates using the light scores, and calculates

\* sum of full scores for those. Divides that by sum of `k` top full scores,

\* overall, to get a "score recall".

\*/

case class SumScoreRecallForTopLight(k: Int) extends EarlybirdEvaluationMetric {

override val name: String = s"sum\_score\_recall\_for\_top\_light\_$k"

override def apply(candidates: Seq[CandidateRecord]): Option[Double] = {

if (candidates.size <= k) {

None

} else {

val sumFullScoresForTopLight = candidates.sortBy(-\_.earlyScore).take(k).map(\_.fullScore).sum

val sumScoresForTopFull = candidates.sortBy(-\_.fullScore).take(k).map(\_.fullScore).sum

Some(sumFullScoresForTopLight / sumScoresForTopFull)

}

}

}

case class HasFewerThanKCandidates(k: Int) extends EarlybirdEvaluationMetric {

override val name: String = s"has\_fewer\_than\_${k}\_candidates"

override def apply(candidates: Seq[CandidateRecord]): Option[Double] =

Some(if (candidates.size <= k) 1.0 else 0.0)

}

case object NumberOfCandidates extends EarlybirdEvaluationMetric {

override val name: String = s"number\_of\_candidates"

override def apply(candidates: Seq[CandidateRecord]): Option[Double] =

Some(candidates.size.toDouble)

}