package com.twitter.product\_mixer.component\_library.candidate\_source.timeline\_scorer

import com.twitter.product\_mixer.core.feature.Feature

import com.twitter.product\_mixer.core.feature.featuremap.FeatureMapBuilder

import com.twitter.product\_mixer.core.functional\_component.candidate\_source.CandidateSourceWithExtractedFeatures

import com.twitter.product\_mixer.core.functional\_component.candidate\_source.CandidatesWithSourceFeatures

import com.twitter.product\_mixer.core.model.common.identifier.CandidateSourceIdentifier

import com.twitter.product\_mixer.core.pipeline.PipelineQuery

import com.twitter.stitch.Stitch

import com.twitter.timelinescorer.common.scoredtweetcandidate.thriftscala.v1

import com.twitter.timelinescorer.common.scoredtweetcandidate.thriftscala.v1.Ancestor

import com.twitter.timelinescorer.common.scoredtweetcandidate.{thriftscala => ct}

import com.twitter.timelinescorer.{thriftscala => t}

import com.twitter.timelineservice.suggests.logging.candidate\_tweet\_source\_id.thriftscala.CandidateTweetSourceId

import javax.inject.Inject

import javax.inject.Singleton

case class ScoredTweetCandidateWithFocalTweet(

candidate: v1.ScoredTweetCandidate,

focalTweetIdOpt: Option[Long])

case object TimelineScorerCandidateSourceSucceededFeature extends Feature[PipelineQuery, Boolean]

@Singleton

class TimelineScorerCandidateSource @Inject() (

timelineScorerClient: t.TimelineScorer.MethodPerEndpoint)

extends CandidateSourceWithExtractedFeatures[

t.ScoredTweetsRequest,

ScoredTweetCandidateWithFocalTweet

] {

override val identifier: CandidateSourceIdentifier =

CandidateSourceIdentifier("TimelineScorer")

private val MaxConversationAncestors = 2

override def apply(

request: t.ScoredTweetsRequest

): Stitch[CandidatesWithSourceFeatures[ScoredTweetCandidateWithFocalTweet]] = {

Stitch

.callFuture(timelineScorerClient.getScoredTweets(request))

.map { response =>

val scoredTweetsOpt = response match {

case t.ScoredTweetsResponse.V1(v1) => v1.scoredTweets

case t.ScoredTweetsResponse.UnknownUnionField(field) =>

throw new UnsupportedOperationException(s"Unknown response type: ${field.field.name}")

}

val scoredTweets = scoredTweetsOpt.getOrElse(Seq.empty)

val allAncestors = scoredTweets.flatMap {

case ct.ScoredTweetCandidate.V1(v1) if isEligibleReply(v1) =>

v1.ancestors.get.map(\_.tweetId)

case \_ => Seq.empty

}.toSet

// Remove tweets within ancestor list of other tweets to avoid serving duplicates

val keptTweets = scoredTweets.collect {

case ct.ScoredTweetCandidate.V1(v1) if !allAncestors.contains(originalTweetId(v1)) => v1

}

// Add parent and root tweet for eligible reply focal tweets

val candidates = keptTweets

.flatMap {

case v1 if isEligibleReply(v1) =>

val ancestors = v1.ancestors.get

val focalTweetId = v1.tweetId

// Include root tweet if the conversation has atleast 2 ancestors

val optionallyIncludedRootTweet = if (ancestors.size >= MaxConversationAncestors) {

val rootTweet = toScoredTweetCandidateFromAncestor(

ancestor = ancestors.last,

inReplyToTweetId = None,

conversationId = v1.conversationId,

ancestors = None,

candidateTweetSourceId = v1.candidateTweetSourceId

)

Seq((rootTweet, Some(v1)))

} else Seq.empty

/\*\*

\* Setting the in-reply-to tweet id on the immediate parent, if one exists,

\* helps ensure tweet type metrics correctly distinguish roots from non-roots.

\*/

val inReplyToTweetId = ancestors.tail.headOption.map(\_.tweetId)

val parentAncestor = toScoredTweetCandidateFromAncestor(

ancestor = ancestors.head,

inReplyToTweetId = inReplyToTweetId,

conversationId = v1.conversationId,

ancestors = Some(ancestors.tail),

candidateTweetSourceId = v1.candidateTweetSourceId

)

optionallyIncludedRootTweet ++

Seq((parentAncestor, Some(v1)), (v1, Some(v1)))

case any => Seq((any, None)) // Set focalTweetId to None if not eligible for convo

}

/\*\*

\* Dedup each tweet keeping the one with highest scored Focal Tweet

\* Focal Tweet ID != the Conversation ID, which is set to the root of the conversation

\* Focal Tweet ID will be defined for tweets with ancestors that should be

\* in conversation modules and None for standalone tweets.

\*/

val sortedDedupedCandidates = candidates

.groupBy { case (v1, \_) => v1.tweetId }

.mapValues { group =>

val (candidate, focalTweetOpt) = group.maxBy {

case (\_, Some(focal)) => focal.score

case (\_, None) => 0

}

ScoredTweetCandidateWithFocalTweet(candidate, focalTweetOpt.map(focal => focal.tweetId))

}.values.toSeq.sortBy(\_.candidate.tweetId)

CandidatesWithSourceFeatures(

candidates = sortedDedupedCandidates,

features = FeatureMapBuilder()

.add(TimelineScorerCandidateSourceSucceededFeature, true)

.build()

)

}

}

private def isEligibleReply(candidate: ct.ScoredTweetCandidateAliases.V1Alias): Boolean = {

candidate.inReplyToTweetId.nonEmpty &&

!candidate.isRetweet.getOrElse(false) &&

candidate.ancestors.exists(\_.nonEmpty)

}

/\*\*

\* If we have a retweet, get the source tweet id.

\* If it is not a retweet, get the regular tweet id.

\*/

private def originalTweetId(candidate: ct.ScoredTweetCandidateAliases.V1Alias): Long = {

candidate.sourceTweetId.getOrElse(candidate.tweetId)

}

private def toScoredTweetCandidateFromAncestor(

ancestor: Ancestor,

inReplyToTweetId: Option[Long],

conversationId: Option[Long],

ancestors: Option[Seq[Ancestor]],

candidateTweetSourceId: Option[CandidateTweetSourceId]

): ct.ScoredTweetCandidateAliases.V1Alias = {

ct.v1.ScoredTweetCandidate(

tweetId = ancestor.tweetId,

authorId = ancestor.userId.getOrElse(0L),

score = 0.0,

isAncestorCandidate = Some(true),

inReplyToTweetId = inReplyToTweetId,

conversationId = conversationId,

ancestors = ancestors,

candidateTweetSourceId = candidateTweetSourceId

)

}

}