package com.twitter.home\_mixer.functional\_component.scorer

import com.twitter.conversions.DurationOps.\_

import com.twitter.home\_mixer.model.HomeFeatures.AuthorIdFeature

import com.twitter.home\_mixer.model.HomeFeatures.FeedbackHistoryFeature

import com.twitter.home\_mixer.model.HomeFeatures.IsRetweetFeature

import com.twitter.home\_mixer.model.HomeFeatures.SGSValidFollowedByUserIdsFeature

import com.twitter.home\_mixer.model.HomeFeatures.SGSValidLikedByUserIdsFeature

import com.twitter.home\_mixer.model.HomeFeatures.ScoreFeature

import com.twitter.home\_mixer.util.CandidatesUtil

import com.twitter.product\_mixer.component\_library.model.candidate.TweetCandidate

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

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

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

import com.twitter.product\_mixer.core.functional\_component.scorer.Scorer

import com.twitter.product\_mixer.core.model.common.CandidateWithFeatures

import com.twitter.product\_mixer.core.model.common.Conditionally

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

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

import com.twitter.stitch.Stitch

import com.twitter.timelines.common.{thriftscala => tl}

import com.twitter.timelineservice.model.FeedbackEntry

import com.twitter.timelineservice.{thriftscala => tls}

import com.twitter.util.Time

import scala.collection.mutable

object FeedbackFatigueScorer

extends Scorer[PipelineQuery, TweetCandidate]

with Conditionally[PipelineQuery] {

override val identifier: ScorerIdentifier = ScorerIdentifier("FeedbackFatigue")

override def features: Set[Feature[\_, \_]] = Set(ScoreFeature)

override def onlyIf(query: PipelineQuery): Boolean =

query.features.exists(\_.getOrElse(FeedbackHistoryFeature, Seq.empty).nonEmpty)

val DurationForFiltering = 14.days

val DurationForDiscounting = 140.days

private val ScoreMultiplierLowerBound = 0.2

private val ScoreMultiplierUpperBound = 1.0

private val ScoreMultiplierIncrementsCount = 4

private val ScoreMultiplierIncrement =

(ScoreMultiplierUpperBound - ScoreMultiplierLowerBound) / ScoreMultiplierIncrementsCount

private val ScoreMultiplierIncrementDurationInDays =

DurationForDiscounting.inDays / ScoreMultiplierIncrementsCount.toDouble

override def apply(

query: PipelineQuery,

candidates: Seq[CandidateWithFeatures[TweetCandidate]]

): Stitch[Seq[FeatureMap]] = {

val feedbackEntriesByEngagementType =

query.features

.getOrElse(FeatureMap.empty).getOrElse(FeedbackHistoryFeature, Seq.empty)

.filter { entry =>

val timeSinceFeedback = query.queryTime.minus(entry.timestamp)

timeSinceFeedback < DurationForFiltering + DurationForDiscounting &&

entry.feedbackType == tls.FeedbackType.SeeFewer

}.groupBy(\_.engagementType)

val authorsToDiscount =

getUserDiscounts(

query.queryTime,

feedbackEntriesByEngagementType.getOrElse(tls.FeedbackEngagementType.Tweet, Seq.empty))

val likersToDiscount =

getUserDiscounts(

query.queryTime,

feedbackEntriesByEngagementType.getOrElse(tls.FeedbackEngagementType.Like, Seq.empty))

val followersToDiscount =

getUserDiscounts(

query.queryTime,

feedbackEntriesByEngagementType.getOrElse(tls.FeedbackEngagementType.Follow, Seq.empty))

val retweetersToDiscount =

getUserDiscounts(

query.queryTime,

feedbackEntriesByEngagementType.getOrElse(tls.FeedbackEngagementType.Retweet, Seq.empty))

val featureMaps = candidates.map { candidate =>

val multiplier = getScoreMultiplier(

candidate,

authorsToDiscount,

likersToDiscount,

followersToDiscount,

retweetersToDiscount

)

val score = candidate.features.getOrElse(ScoreFeature, None)

FeatureMapBuilder().add(ScoreFeature, score.map(\_ \* multiplier)).build()

}

Stitch.value(featureMaps)

}

def getScoreMultiplier(

candidate: CandidateWithFeatures[TweetCandidate],

authorsToDiscount: Map[Long, Double],

likersToDiscount: Map[Long, Double],

followersToDiscount: Map[Long, Double],

retweetersToDiscount: Map[Long, Double],

): Double = {

val originalAuthorId =

CandidatesUtil.getOriginalAuthorId(candidate.features).getOrElse(0L)

val originalAuthorMultiplier = authorsToDiscount.getOrElse(originalAuthorId, 1.0)

val likers = candidate.features.getOrElse(SGSValidLikedByUserIdsFeature, Seq.empty)

val likerMultipliers = likers.flatMap(likersToDiscount.get)

val likerMultiplier =

if (likerMultipliers.nonEmpty && likers.size == likerMultipliers.size)

likerMultipliers.max

else 1.0

val followers = candidate.features.getOrElse(SGSValidFollowedByUserIdsFeature, Seq.empty)

val followerMultipliers = followers.flatMap(followersToDiscount.get)

val followerMultiplier =

if (followerMultipliers.nonEmpty && followers.size == followerMultipliers.size &&

likers.isEmpty)

followerMultipliers.max

else 1.0

val authorId = candidate.features.getOrElse(AuthorIdFeature, None).getOrElse(0L)

val retweeterMultiplier =

if (candidate.features.getOrElse(IsRetweetFeature, false))

retweetersToDiscount.getOrElse(authorId, 1.0)

else 1.0

originalAuthorMultiplier \* likerMultiplier \* followerMultiplier \* retweeterMultiplier

}

def getUserDiscounts(

queryTime: Time,

feedbackEntries: Seq[FeedbackEntry],

): Map[Long, Double] = {

val userDiscounts = mutable.Map[Long, Double]()

feedbackEntries

.collect {

case FeedbackEntry(\_, \_, tl.FeedbackEntity.UserId(userId), timestamp, \_) =>

val timeSinceFeedback = queryTime.minus(timestamp)

val timeSinceDiscounting = timeSinceFeedback - DurationForFiltering

val multiplier = ((timeSinceDiscounting.inDays / ScoreMultiplierIncrementDurationInDays)

\* ScoreMultiplierIncrement + ScoreMultiplierLowerBound)

userDiscounts.update(userId, multiplier)

}

userDiscounts.toMap

}

}