package com.twitter.frigate.pushservice.predicate.magic\_fanout

import com.twitter.eventdetection.event\_context.util.SimClustersUtil

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.magic\_events.thriftscala.\_

import com.twitter.frigate.pushservice.model.MagicFanoutEventPushCandidate

import com.twitter.frigate.pushservice.model.MagicFanoutNewsEventPushCandidate

import com.twitter.frigate.pushservice.model.MagicFanoutProductLaunchPushCandidate

import com.twitter.frigate.pushservice.params.PushFeatureSwitchParams

import com.twitter.frigate.thriftscala.CommonRecommendationType

import com.twitter.simclusters\_v2.common.SimClustersEmbedding

import com.twitter.simclusters\_v2.thriftscala.EmbeddingType

import com.twitter.simclusters\_v2.thriftscala.ModelVersion

import com.twitter.simclusters\_v2.thriftscala.SimClustersEmbeddingId

import com.twitter.simclusters\_v2.thriftscala.{SimClustersEmbedding => ThriftSimClustersEmbedding}

import com.twitter.util.Future

object MagicFanoutPredicatesUtil {

val UttDomain: Long = 0L

type DomainId = Long

type EntityId = Long

val BroadCategoryTag = "utt:broad\_category"

val UgmMomentTag = "MMTS.isUGMMoment"

val TopKSimClustersCount = 50

case class SimClusterScores(simClusterScoreVector: Map[Int, Double]) {

def dotProduct(other: SimClusterScores): Double = {

simClusterScoreVector

.map {

case (clusterId, score) => other.simClusterScoreVector.getOrElse(clusterId, 0.0) \* score

}.foldLeft(0.0) { \_ + \_ }

}

def norm(): Double = {

val sumOfSquares: Double = simClusterScoreVector

.map {

case (clusterId, score) => score \* score

}.foldLeft(0.0)(\_ + \_)

scala.math.sqrt(sumOfSquares)

}

def normedDotProduct(other: SimClusterScores, normalizer: SimClusterScores): Double = {

val denominator = normalizer.norm()

val score = dotProduct(other)

if (denominator != 0.0) {

score / denominator

} else {

score

}

}

}

private def isSemanticCoreEntityBroad(

semanticCoreEntityTags: Map[(DomainId, EntityId), Set[String]],

scEntityId: SemanticCoreID

): Boolean = {

semanticCoreEntityTags

.getOrElse((scEntityId.domainId, scEntityId.entityId), Set.empty).contains(BroadCategoryTag)

}

def isInCountryList(accountCountryCode: String, locales: Seq[String]): Boolean = {

locales.map(\_.toLowerCase).contains(accountCountryCode.toLowerCase)

}

/\*\*

\* Boolean check of if a MagicFanout is high priority push

\*/

def checkIfHighPriorityNewsEventForCandidate(

candidate: MagicFanoutNewsEventPushCandidate

): Future[Boolean] = {

candidate.isHighPriorityEvent.map { isHighPriority =>

isHighPriority && (candidate.target.params(PushFeatureSwitchParams.EnableHighPriorityPush))

}

}

/\*\*

\* Boolean check of if a MagicFanout event is high priority push

\*/

def checkIfHighPriorityEventForCandidate(

candidate: MagicFanoutEventPushCandidate

): Future[Boolean] = {

candidate.isHighPriorityEvent.map { isHighPriority =>

candidate.commonRecType match {

case CommonRecommendationType.MagicFanoutSportsEvent =>

isHighPriority && (candidate.target.params(

PushFeatureSwitchParams.EnableHighPrioritySportsPush))

case \_ => false

}

}

}

/\*\*

\* Boolean check if to skip target blue verified

\*/

def shouldSkipBlueVerifiedCheckForCandidate(

candidate: MagicFanoutProductLaunchPushCandidate

): Future[Boolean] =

Future.value(

candidate.target.params(PushFeatureSwitchParams.DisableIsTargetBlueVerifiedPredicate))

/\*\*

\* Boolean check if to skip target is legacy verified

\*/

def shouldSkipLegacyVerifiedCheckForCandidate(

candidate: MagicFanoutProductLaunchPushCandidate

): Future[Boolean] =

Future.value(

candidate.target.params(PushFeatureSwitchParams.DisableIsTargetLegacyVerifiedPredicate))

def shouldSkipSuperFollowCreatorCheckForCandidate(

candidate: MagicFanoutProductLaunchPushCandidate

): Future[Boolean] =

Future.value(

!candidate.target.params(PushFeatureSwitchParams.EnableIsTargetSuperFollowCreatorPredicate))

/\*\*

\* Boolean check of if a reason of a MagicFanout is higher than the rank threshold of an event

\*/

def checkIfErgScEntityReasonMeetsThreshold(

rankThreshold: Int,

reason: MagicEventsReason,

): Boolean = {

reason.reason match {

case TargetID.SemanticCoreID(scEntityId: SemanticCoreID) =>

reason.rank match {

case Some(rank) => rank < rankThreshold

case \_ => false

}

case \_ => false

}

}

/\*\*

\* Check if MagicEventsReasons contains a reason that matches the thresholdw

\*/

def checkIfValidErgScEntityReasonExists(

magicEventsReasons: Option[Seq[MagicEventsReason]],

rankThreshold: Int

)(

implicit stats: StatsReceiver

): Boolean = {

magicEventsReasons match {

case Some(reasons) if reasons.exists(\_.isNewUser.contains(true)) => true

case Some(reasons) =>

reasons.exists { reason =>

reason.source.contains(ReasonSource.ErgShortTermInterestSemanticCore) &&

checkIfErgScEntityReasonMeetsThreshold(

rankThreshold,

reason

)

}

case \_ => false

}

}

/\*\*

\* Get event simcluster vector from event context

\*/

def getEventSimClusterVector(

simClustersEmbeddingOption: Option[Map[SimClustersEmbeddingId, ThriftSimClustersEmbedding]],

embeddingMapKey: (ModelVersion, EmbeddingType),

topKSimClustersCount: Int

): Option[SimClusterScores] = {

simClustersEmbeddingOption.map { thriftSimClustersEmbeddings =>

val simClustersEmbeddings: Map[SimClustersEmbeddingId, SimClustersEmbedding] =

thriftSimClustersEmbeddings.map {

case (simClustersEmbeddingId, simClustersEmbeddingValue) =>

(simClustersEmbeddingId, SimClustersEmbedding(simClustersEmbeddingValue))

}.toMap

val emptySeq = Seq[(Int, Double)]()

val simClusterScoreTuple: Map[(ModelVersion, EmbeddingType), Seq[(Int, Double)]] =

SimClustersUtil

.getMaxTopKTweetSimClusters(simClustersEmbeddings, topKSimClustersCount)

SimClusterScores(simClusterScoreTuple.getOrElse(embeddingMapKey, emptySeq).toMap)

}

}

/\*\*

\* Get user simcluster vector magic events reasons

\*/

def getUserSimClusterVector(

magicEventsReasonsOpt: Option[Seq[MagicEventsReason]]

): Option[SimClusterScores] = {

magicEventsReasonsOpt.map { magicEventsReasons: Seq[MagicEventsReason] =>

val reasons: Seq[(Int, Double)] = magicEventsReasons.flatMap { reason =>

reason.reason match {

case TargetID.SimClusterID(simClusterId: SimClusterID) =>

Some((simClusterId.clusterId, reason.score.getOrElse(0.0)))

case \_ =>

None

}

}

SimClusterScores(reasons.toMap)

}

}

def reasonsContainGeoTarget(reasons: Seq[MagicEventsReason]): Boolean = {

reasons.exists { reason =>

val isGeoGraphSource = reason.source.contains(ReasonSource.GeoGraph)

reason.reason match {

case TargetID.PlaceID(\_) if isGeoGraphSource => true

case \_ => false

}

}

}

def geoPlaceIdsFromReasons(reasons: Seq[MagicEventsReason]): Set[Long] = {

reasons.flatMap { reason =>

val isGeoGraphSource = reason.source.contains(ReasonSource.GeoGraph)

reason.reason match {

case TargetID.PlaceID(PlaceID(id)) if isGeoGraphSource => Some(id)

case \_ => None

}

}.toSet

}

}