package com.twitter.frigate.pushservice.util

import com.twitter.contentrecommender.thriftscala.DisplayLocation

import com.twitter.finagle.stats.Stat

import com.twitter.frigate.common.base.TargetUser

import com.twitter.frigate.common.predicate.CommonOutNetworkTweetCandidatesSourcePredicates.authorNotBeingFollowedPredicate

import com.twitter.frigate.common.store.interests.InterestsLookupRequestWithContext

import com.twitter.frigate.pushservice.model.PushTypes.Target

import com.twitter.frigate.pushservice.model.PushTypes

import com.twitter.frigate.pushservice.store.UttEntityHydrationQuery

import com.twitter.frigate.pushservice.store.UttEntityHydrationStore

import com.twitter.hermit.predicate.Predicate

import com.twitter.hermit.predicate.socialgraph.RelationEdge

import com.twitter.interests.thriftscala.InterestRelationType

import com.twitter.interests.thriftscala.InterestRelationship

import com.twitter.interests.thriftscala.InterestedInInterestLookupContext

import com.twitter.interests.thriftscala.InterestedInInterestModel

import com.twitter.interests.thriftscala.ProductId

import com.twitter.interests.thriftscala.UserInterest

import com.twitter.interests.thriftscala.UserInterestData

import com.twitter.interests.thriftscala.UserInterests

import com.twitter.interests.thriftscala.{TopicListingViewerContext => TopicListingViewerContextCR}

import com.twitter.stitch.tweetypie.TweetyPie.TweetyPieResult

import com.twitter.storehaus.ReadableStore

import com.twitter.timelines.configapi.Param

import com.twitter.topiclisting.TopicListingViewerContext

import com.twitter.topiclisting.utt.LocalizedEntity

import com.twitter.tsp.thriftscala.TopicListingSetting

import com.twitter.tsp.thriftscala.TopicSocialProofRequest

import com.twitter.tsp.thriftscala.TopicSocialProofResponse

import com.twitter.tsp.thriftscala.TopicWithScore

import com.twitter.util.Future

import scala.collection.Map

case class TweetWithTopicProof(

tweetId: Long,

topicId: Long,

authorId: Option[Long],

score: Double,

tweetyPieResult: TweetyPieResult,

topicListingSetting: String,

algorithmCR: Option[String],

isOON: Boolean)

object TopicsUtil {

/\*\*

\* Obtains the Localized Entities for the provided SC Entity IDs

\* @param target The target user for which we're obtaining candidates

\* @param semanticCoreEntityIds The seq. of entity ids for which we would like to obtain the Localized Entities

\* @param uttEntityHydrationStore Store to query the actual LocalizedEntities

\* @return A Future Map consisting of the entity id as the key and LocalizedEntity as the value

\*/

def getLocalizedEntityMap(

target: Target,

semanticCoreEntityIds: Set[Long],

uttEntityHydrationStore: UttEntityHydrationStore

): Future[Map[Long, LocalizedEntity]] = {

buildTopicListingViewerContext(target)

.flatMap { topicListingViewerContext =>

val query = UttEntityHydrationQuery(topicListingViewerContext, semanticCoreEntityIds.toSeq)

val localizedTopicEntitiesFut =

uttEntityHydrationStore.getLocalizedTopicEntities(query).map(\_.flatten)

localizedTopicEntitiesFut.map { localizedTopicEntities =>

localizedTopicEntities.map { localizedTopicEntity =>

localizedTopicEntity.entityId -> localizedTopicEntity

}.toMap

}

}

}

/\*\*

\* Fetch explict followed interests i.e Topics for targetUser

\*

\* @param targetUser: [[Target]] object representing a user eligible for MagicRecs notification

\* @return: list of all Topics(Interests) Followed by targetUser

\*/

def getTopicsFollowedByUser(

targetUser: Target,

interestsWithLookupContextStore: ReadableStore[

InterestsLookupRequestWithContext,

UserInterests

],

followedTopicsStats: Stat

): Future[Option[Seq[UserInterest]]] = {

buildTopicListingViewerContext(targetUser).flatMap { topicListingViewerContext =>

// explicit interests relation query

val explicitInterestsLookupRequest = InterestsLookupRequestWithContext(

targetUser.targetId,

Some(

InterestedInInterestLookupContext(

explicitContext = None,

inferredContext = None,

productId = Some(ProductId.Followable),

topicListingViewerContext = Some(topicListingViewerContext.toThrift),

disableExplicit = None,

disableImplicit = Some(true)

)

)

)

// filter explicit follow relationships from response

interestsWithLookupContextStore.get(explicitInterestsLookupRequest).map {

\_.flatMap { userInterests =>

val followedTopics = userInterests.interests.map {

\_.filter {

case UserInterest(\_, Some(interestData)) =>

interestData match {

case UserInterestData.InterestedIn(interestedIn) =>

interestedIn.exists {

case InterestedInInterestModel.ExplicitModel(explicitModel) =>

explicitModel match {

case InterestRelationship.V1(v1) =>

v1.relation == InterestRelationType.Followed

case \_ => false

}

case \_ => false

}

case \_ => false

}

case \_ => false // interestData unavailable

}

}

followedTopicsStats.add(followedTopics.getOrElse(Seq.empty[UserInterest]).size)

followedTopics

}

}

}

}

/\*\*

\*

\* @param target : [[Target]] object respresenting MagicRecs user

\*

\* @return: [[TopicListingViewerContext]] for querying topics

\*/

def buildTopicListingViewerContext(target: Target): Future[TopicListingViewerContext] = {

Future.join(target.inferredUserDeviceLanguage, target.countryCode, target.targetUser).map {

case (inferredLanguage, countryCode, userInfo) =>

TopicListingViewerContext(

userId = Some(target.targetId),

guestId = None,

deviceId = None,

clientApplicationId = None,

userAgent = None,

languageCode = inferredLanguage,

countryCode = countryCode,

userRoles = userInfo.flatMap(\_.roles.map(\_.roles.toSet))

)

}

}

/\*\*

\*

\* @param target : [[Target]] object respresenting MagicRecs user

\*

\* @return: [[TopicListingViewerContext]] for querying topics

\*/

def buildTopicListingViewerContextForCR(target: Target): Future[TopicListingViewerContextCR] = {

TopicsUtil.buildTopicListingViewerContext(target).map(\_.toThrift)

}

/\*\*

\*

\* @param target : [[Target]] object respresenting MagicRecs user

\* @param tweets : [[Seq[TweetyPieResult]]] object representing Tweets to get TSP for

\* @param topicSocialProofServiceStore: [[ReadableStore[TopicSocialProofRequest, TopicSocialProofResponse]]]

\* @param edgeStore: [[ReadableStore[RelationEdge, Boolean]]]]

\*

\* @return: [[Future[Seq[TweetWithTopicProof]]]] Tweets with topic proof

\*/

def getTopicSocialProofs(

inputTarget: Target,

tweets: Seq[TweetyPieResult],

topicSocialProofServiceStore: ReadableStore[TopicSocialProofRequest, TopicSocialProofResponse],

edgeStore: ReadableStore[RelationEdge, Boolean],

scoreThresholdParam: Param[Double]

): Future[Seq[TweetWithTopicProof]] = {

buildTopicListingViewerContextForCR(inputTarget).flatMap { topicListingContext =>

val tweetIds: Set[Long] = tweets.map(\_.tweet.id).toSet

val tweetIdsToTweetyPie = tweets.map(tp => tp.tweet.id -> tp).toMap

val topicSocialProofRequest =

TopicSocialProofRequest(

inputTarget.targetId,

tweetIds,

DisplayLocation.MagicRecsRecommendTopicTweets,

TopicListingSetting.Followable,

topicListingContext)

topicSocialProofServiceStore

.get(topicSocialProofRequest).flatMap {

case Some(topicSocialProofResponse) =>

val topicProofCandidates = topicSocialProofResponse.socialProofs.collect {

case (tweetId, topicsWithScore)

if topicsWithScore.nonEmpty && topicsWithScore

.maxBy(\_.score).score >= inputTarget

.params(scoreThresholdParam) =>

// Get the topic with max score if there are any topics returned

val topicWithScore = topicsWithScore.maxBy(\_.score)

TweetWithTopicProof(

tweetId,

topicWithScore.topicId,

tweetIdsToTweetyPie(tweetId).tweet.coreData.map(\_.userId),

topicWithScore.score,

tweetIdsToTweetyPie(tweetId),

topicWithScore.topicFollowType.map(\_.name).getOrElse(""),

topicWithScore.algorithmType.map(\_.name),

isOON = true

)

}.toSeq

hydrateTopicProofCandidatesWithEdgeStore(inputTarget, topicProofCandidates, edgeStore)

case \_ => Future.value(Seq.empty[TweetWithTopicProof])

}

}

}

/\*\*

\* Obtain TopicWithScores for provided tweet candidates and target

\* @param target target user

\* @param Tweets tweet candidates represented in a (tweetId, TweetyPieResult) map

\* @param topicSocialProofServiceStore store to query topic social proof

\* @param enableTopicAnnotation whether to enable topic annotation

\* @param topicScoreThreshold threshold for topic score

\* @return a (tweetId, TopicWithScore) map where the topic with highest topic score (if exists) is chosen

\*/

def getTopicsWithScoreMap(

target: PushTypes.Target,

Tweets: Map[Long, Option[TweetyPieResult]],

topicSocialProofServiceStore: ReadableStore[TopicSocialProofRequest, TopicSocialProofResponse],

enableTopicAnnotation: Boolean,

topicScoreThreshold: Double

): Future[Option[Map[Long, TopicWithScore]]] = {

if (enableTopicAnnotation) {

TopicsUtil

.buildTopicListingViewerContextForCR(target).flatMap { topicListingContext =>

val tweetIds = Tweets.keySet

val topicSocialProofRequest =

TopicSocialProofRequest(

target.targetId,

tweetIds,

DisplayLocation.MagicRecsRecommendTopicTweets,

TopicListingSetting.Followable,

topicListingContext)

topicSocialProofServiceStore

.get(topicSocialProofRequest).map {

\_.map { topicSocialProofResponse =>

topicSocialProofResponse.socialProofs

.collect {

case (tweetId, topicsWithScore)

if topicsWithScore.nonEmpty && Tweets(tweetId).nonEmpty

&& topicsWithScore.maxBy(\_.score).score >= topicScoreThreshold =>

tweetId -> topicsWithScore.maxBy(\_.score)

}

}

}

}

} else {

Future.None

}

}

/\*\*

\* Obtain LocalizedEntities for provided tweet candidates and target

\* @param target target user

\* @param Tweets tweet candidates represented in a (tweetId, TweetyPieResult) map

\* @param uttEntityHydrationStore store to query the actual LocalizedEntities

\* @param topicSocialProofServiceStore store to query topic social proof

\* @param enableTopicAnnotation whether to enable topic annotation

\* @param topicScoreThreshold threshold for topic score

\* @return a (tweetId, LocalizedEntity Option) Future map that stores Localized Entity (can be empty) for given tweetId

\*/

def getTweetIdLocalizedEntityMap(

target: PushTypes.Target,

Tweets: Map[Long, Option[TweetyPieResult]],

uttEntityHydrationStore: UttEntityHydrationStore,

topicSocialProofServiceStore: ReadableStore[TopicSocialProofRequest, TopicSocialProofResponse],

enableTopicAnnotation: Boolean,

topicScoreThreshold: Double

): Future[Map[Long, Option[LocalizedEntity]]] = {

val topicWithScoreMap = getTopicsWithScoreMap(

target,

Tweets,

topicSocialProofServiceStore,

enableTopicAnnotation,

topicScoreThreshold)

topicWithScoreMap.flatMap { topicWithScores =>

topicWithScores match {

case Some(topics) =>

val topicIds = topics.collect { case (\_, topic) => topic.topicId }.toSet

val LocalizedEntityMapFut =

getLocalizedEntityMap(target, topicIds, uttEntityHydrationStore)

LocalizedEntityMapFut.map { LocalizedEntityMap =>

topics.map {

case (tweetId, topic) =>

tweetId -> LocalizedEntityMap.get(topic.topicId)

}

}

case \_ => Future.value(Map[Long, Option[LocalizedEntity]]())

}

}

}

/\*\*

\* Hydrate TweetWithTopicProof candidates with isOON field info,

\* based on the following relationship between target user and candidate author in edgeStore

\* @return TweetWithTopicProof candidates with isOON field populated

\*/

def hydrateTopicProofCandidatesWithEdgeStore(

inputTarget: TargetUser,

topicProofCandidates: Seq[TweetWithTopicProof],

edgeStore: ReadableStore[RelationEdge, Boolean],

): Future[Seq[TweetWithTopicProof]] = {

// IDs of all authors of TopicProof candidates that are OON with respect to inputTarget

val validOONAuthorIdsFut =

Predicate.filter(

topicProofCandidates.flatMap(\_.authorId).distinct,

authorNotBeingFollowedPredicate(inputTarget, edgeStore))

validOONAuthorIdsFut.map { validOONAuthorIds =>

topicProofCandidates.map(candidate => {

candidate.copy(isOON =

candidate.authorId.isDefined && validOONAuthorIds.contains(candidate.authorId.get))

})

}

}

}