package com.twitter.frigate.pushservice.adaptor

import com.twitter.finagle.stats.Stat

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.base.CandidateSource

import com.twitter.frigate.common.base.CandidateSourceEligible

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

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

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

import com.twitter.frigate.pushservice.params.HighQualityCandidateGroupEnum

import com.twitter.frigate.pushservice.params.HighQualityCandidateGroupEnum.\_

import com.twitter.frigate.pushservice.params.PushConstants.targetUserAgeFeatureName

import com.twitter.frigate.pushservice.params.PushConstants.targetUserPreferredLanguage

import com.twitter.frigate.pushservice.params.{PushFeatureSwitchParams => FS}

import com.twitter.frigate.pushservice.predicate.TargetPredicates

import com.twitter.frigate.pushservice.util.MediaCRT

import com.twitter.frigate.pushservice.util.PushAdaptorUtil

import com.twitter.frigate.pushservice.util.PushDeviceUtil

import com.twitter.frigate.pushservice.util.TopicsUtil

import com.twitter.frigate.thriftscala.CommonRecommendationType

import com.twitter.interests.thriftscala.InterestId.SemanticCore

import com.twitter.interests.thriftscala.UserInterests

import com.twitter.language.normalization.UserDisplayLanguage

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

import com.twitter.storehaus.ReadableStore

import com.twitter.trends.trip\_v1.trip\_tweets.thriftscala.TripDomain

import com.twitter.trends.trip\_v1.trip\_tweets.thriftscala.TripTweet

import com.twitter.trends.trip\_v1.trip\_tweets.thriftscala.TripTweets

import com.twitter.util.Future

object HighQualityTweetsHelper {

def getFollowedTopics(

target: Target,

interestsWithLookupContextStore: ReadableStore[

InterestsLookupRequestWithContext,

UserInterests

],

followedTopicsStats: Stat

): Future[Seq[Long]] = {

TopicsUtil

.getTopicsFollowedByUser(target, interestsWithLookupContextStore, followedTopicsStats).map {

userInterestsOpt =>

val userInterests = userInterestsOpt.getOrElse(Seq.empty)

val extractedTopicIds = userInterests.flatMap {

\_.interestId match {

case SemanticCore(semanticCore) => Some(semanticCore.id)

case \_ => None

}

}

extractedTopicIds

}

}

def getTripQueries(

target: Target,

enabledGroups: Set[HighQualityCandidateGroupEnum.Value],

interestsWithLookupContextStore: ReadableStore[

InterestsLookupRequestWithContext,

UserInterests

],

sourceIds: Seq[String],

stat: Stat

): Future[Set[TripDomain]] = {

val followedTopicIdsSetFut: Future[Set[Long]] = if (enabledGroups.contains(Topic)) {

getFollowedTopics(target, interestsWithLookupContextStore, stat).map(topicIds =>

topicIds.toSet)

} else {

Future.value(Set.empty)

}

Future

.join(target.featureMap, target.inferredUserDeviceLanguage, followedTopicIdsSetFut).map {

case (

featureMap,

deviceLanguageOpt,

followedTopicIds

) =>

val ageBucketOpt = if (enabledGroups.contains(AgeBucket)) {

featureMap.categoricalFeatures.get(targetUserAgeFeatureName)

} else {

None

}

val languageOptions: Set[Option[String]] = if (enabledGroups.contains(Language)) {

val userPreferredLanguages = featureMap.sparseBinaryFeatures

.getOrElse(targetUserPreferredLanguage, Set.empty[String])

if (userPreferredLanguages.nonEmpty) {

userPreferredLanguages.map(lang => Some(UserDisplayLanguage.toTweetLanguage(lang)))

} else {

Set(deviceLanguageOpt.map(UserDisplayLanguage.toTweetLanguage))

}

} else Set(None)

val followedTopicOptions: Set[Option[Long]] = if (followedTopicIds.nonEmpty) {

followedTopicIds.map(topic => Some(topic))

} else Set(None)

val tripQueries = followedTopicOptions.flatMap { topicOption =>

languageOptions.flatMap { languageOption =>

sourceIds.map { sourceId =>

TripDomain(

sourceId = sourceId,

language = languageOption,

placeId = None,

topicId = topicOption,

gender = None,

ageBucket = ageBucketOpt

)

}

}

}

tripQueries

}

}

}

case class HighQualityTweetsAdaptor(

tripTweetCandidateStore: ReadableStore[TripDomain, TripTweets],

interestsWithLookupContextStore: ReadableStore[InterestsLookupRequestWithContext, UserInterests],

tweetyPieStore: ReadableStore[Long, TweetyPieResult],

tweetyPieStoreNoVF: ReadableStore[Long, TweetyPieResult],

globalStats: StatsReceiver)

extends CandidateSource[Target, RawCandidate]

with CandidateSourceEligible[Target, RawCandidate] {

override def name: String = this.getClass.getSimpleName

private val stats = globalStats.scope("HighQualityCandidateAdaptor")

private val followedTopicsStats = stats.stat("followed\_topics")

private val missingResponseCounter = stats.counter("missing\_respond\_counter")

private val crtFatigueCounter = stats.counter("fatigue\_by\_crt")

private val fallbackRequestsCounter = stats.counter("fallback\_requests")

override def isCandidateSourceAvailable(target: Target): Future[Boolean] = {

PushDeviceUtil.isRecommendationsEligible(target).map {

\_ && target.params(FS.HighQualityCandidatesEnableCandidateSource)

}

}

private val highQualityCandidateFrequencyPredicate = {

TargetPredicates

.pushRecTypeFatiguePredicate(

CommonRecommendationType.TripHqTweet,

FS.HighQualityTweetsPushInterval,

FS.MaxHighQualityTweetsPushGivenInterval,

stats

)

}

private def getTripCandidatesStrato(

target: Target

): Future[Map[Long, Set[TripDomain]]] = {

val tripQueriesF: Future[Set[TripDomain]] = HighQualityTweetsHelper.getTripQueries(

target = target,

enabledGroups = target.params(FS.HighQualityCandidatesEnableGroups).toSet,

interestsWithLookupContextStore = interestsWithLookupContextStore,

sourceIds = target.params(FS.TripTweetCandidateSourceIds),

stat = followedTopicsStats

)

lazy val fallbackTripQueriesFut: Future[Set[TripDomain]] =

if (target.params(FS.HighQualityCandidatesEnableFallback))

HighQualityTweetsHelper.getTripQueries(

target = target,

enabledGroups = target.params(FS.HighQualityCandidatesFallbackEnabledGroups).toSet,

interestsWithLookupContextStore = interestsWithLookupContextStore,

sourceIds = target.params(FS.HighQualityCandidatesFallbackSourceIds),

stat = followedTopicsStats

)

else Future.value(Set.empty)

val initialTweetsFut: Future[Map[TripDomain, Seq[TripTweet]]] = tripQueriesF.flatMap {

tripQueries => getTripTweetsByDomains(tripQueries)

}

val tweetsByDomainFut: Future[Map[TripDomain, Seq[TripTweet]]] =

if (target.params(FS.HighQualityCandidatesEnableFallback)) {

initialTweetsFut.flatMap { candidates =>

val minCandidatesForFallback: Int =

target.params(FS.HighQualityCandidatesMinNumOfCandidatesToFallback)

val validCandidates = candidates.filter(\_.\_2.size >= minCandidatesForFallback)

if (validCandidates.nonEmpty) {

Future.value(validCandidates)

} else {

fallbackTripQueriesFut.flatMap { fallbackTripDomains =>

fallbackRequestsCounter.incr(fallbackTripDomains.size)

getTripTweetsByDomains(fallbackTripDomains)

}

}

}

} else {

initialTweetsFut

}

val numOfCandidates: Int = target.params(FS.HighQualityCandidatesNumberOfCandidates)

tweetsByDomainFut.map(tweetsByDomain => reformatDomainTweetMap(tweetsByDomain, numOfCandidates))

}

private def getTripTweetsByDomains(

tripQueries: Set[TripDomain]

): Future[Map[TripDomain, Seq[TripTweet]]] = {

Future.collect(tripTweetCandidateStore.multiGet(tripQueries)).map { response =>

response

.filter(p => p.\_2.exists(\_.tweets.nonEmpty))

.mapValues(\_.map(\_.tweets).getOrElse(Seq.empty))

}

}

private def reformatDomainTweetMap(

tweetsByDomain: Map[TripDomain, Seq[TripTweet]],

numOfCandidates: Int

): Map[Long, Set[TripDomain]] = tweetsByDomain

.flatMap {

case (tripDomain, tripTweets) =>

tripTweets

.sortBy(\_.score)(Ordering[Double].reverse)

.take(numOfCandidates)

.map { tweet => (tweet.tweetId, tripDomain) }

}.groupBy(\_.\_1).mapValues(\_.map(\_.\_2).toSet)

private def buildRawCandidate(

target: Target,

tweetyPieResult: TweetyPieResult,

tripDomain: Option[scala.collection.Set[TripDomain]]

): RawCandidate = {

PushAdaptorUtil.generateOutOfNetworkTweetCandidates(

inputTarget = target,

id = tweetyPieResult.tweet.id,

mediaCRT = MediaCRT(

CommonRecommendationType.TripHqTweet,

CommonRecommendationType.TripHqTweet,

CommonRecommendationType.TripHqTweet

),

result = Some(tweetyPieResult),

tripTweetDomain = tripDomain

)

}

private def getTweetyPieResults(

target: Target,

tweetToTripDomain: Map[Long, Set[TripDomain]]

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

Future.collect((if (target.params(FS.EnableVFInTweetypie)) {

tweetyPieStore

} else {

tweetyPieStoreNoVF

}).multiGet(tweetToTripDomain.keySet))

}

override def get(target: Target): Future[Option[Seq[RawCandidate]]] = {

for {

tweetsToTripDomainMap <- getTripCandidatesStrato(target)

tweetyPieResults <- getTweetyPieResults(target, tweetsToTripDomainMap)

} yield {

val candidates = tweetyPieResults.flatMap {

case (tweetId, tweetyPieResultOpt) =>

tweetyPieResultOpt.map(buildRawCandidate(target, \_, tweetsToTripDomainMap.get(tweetId)))

}

if (candidates.nonEmpty) {

highQualityCandidateFrequencyPredicate(Seq(target))

.map(\_.head)

.map { isTargetFatigueEligible =>

if (isTargetFatigueEligible) Some(candidates)

else {

crtFatigueCounter.incr()

None

}

}

Some(candidates.toSeq)

} else {

missingResponseCounter.incr()

None

}

}

}

}