package com.twitter.frigate.pushservice.adaptor

import com.twitter.finagle.stats.Counter

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.base.TweetCandidate

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

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

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

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

import com.twitter.frigate.pushservice.params.PopGeoTweetVersion

import com.twitter.frigate.pushservice.params.PushParams

import com.twitter.frigate.pushservice.params.TopTweetsForGeoCombination

import com.twitter.frigate.pushservice.params.TopTweetsForGeoRankingFunction

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

import com.twitter.frigate.pushservice.predicate.DiscoverTwitterPredicate

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.thriftscala.CommonRecommendationType

import com.twitter.geoduck.common.thriftscala.{Location => GeoLocation}

import com.twitter.geoduck.service.thriftscala.LocationResponse

import com.twitter.gizmoduck.thriftscala.UserType

import com.twitter.hermit.pop\_geo.thriftscala.PopTweetsInPlace

import com.twitter.recommendation.interests.discovery.core.model.InterestDomain

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

import com.twitter.storehaus.FutureOps

import com.twitter.storehaus.ReadableStore

import com.twitter.util.Future

import com.twitter.util.Time

import scala.collection.Map

case class PlaceTweetScore(place: String, tweetId: Long, score: Double) {

def toTweetScore: (Long, Double) = (tweetId, score)

}

case class TopTweetsByGeoAdaptor(

geoduckStoreV2: ReadableStore[Long, LocationResponse],

softUserGeoLocationStore: ReadableStore[Long, GeoLocation],

topTweetsByGeoStore: ReadableStore[InterestDomain[String], Map[String, List[(Long, Double)]]],

topTweetsByGeoStoreV2: ReadableStore[String, PopTweetsInPlace],

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[this] val stats = globalStats.scope("TopTweetsByGeoAdaptor")

private[this] val noGeohashUserCounter: Counter = stats.counter("users\_with\_no\_geohash\_counter")

private[this] val incomingRequestCounter: Counter = stats.counter("incoming\_request\_counter")

private[this] val incomingLoggedOutRequestCounter: Counter =

stats.counter("incoming\_logged\_out\_request\_counter")

private[this] val loggedOutRawCandidatesCounter =

stats.counter("logged\_out\_raw\_candidates\_counter")

private[this] val emptyLoggedOutRawCandidatesCounter =

stats.counter("logged\_out\_empty\_raw\_candidates")

private[this] val outputTopTweetsByGeoCounter: Stat =

stats.stat("output\_top\_tweets\_by\_geo\_counter")

private[this] val loggedOutPopByGeoV2CandidatesCounter: Counter =

stats.counter("logged\_out\_pop\_by\_geo\_candidates")

private[this] val dormantUsersSince14DaysCounter: Counter =

stats.counter("dormant\_user\_since\_14\_days\_counter")

private[this] val dormantUsersSince30DaysCounter: Counter =

stats.counter("dormant\_user\_since\_30\_days\_counter")

private[this] val nonDormantUsersSince14DaysCounter: Counter =

stats.counter("non\_dormant\_user\_since\_14\_days\_counter")

private[this] val topTweetsByGeoTake100Counter: Counter =

stats.counter("top\_tweets\_by\_geo\_take\_100\_counter")

private[this] val combinationRequestsCounter =

stats.scope("combination\_method\_request\_counter")

private[this] val popGeoTweetVersionCounter =

stats.scope("popgeo\_tweet\_version\_counter")

private[this] val nonReplyTweetsCounter = stats.counter("non\_reply\_tweets")

val MaxGeoHashSize = 4

private def constructKeys(

geohash: Option[String],

accountCountryCode: Option[String],

keyLengths: Seq[Int],

version: PopGeoTweetVersion.Value

): Set[String] = {

val geohashKeys = geohash match {

case Some(hash) => keyLengths.map { version + "\_geohash\_" + hash.take(\_) }

case \_ => Seq.empty

}

val accountCountryCodeKeys =

accountCountryCode.toSeq.map(version + "\_country\_" + \_.toUpperCase)

(geohashKeys ++ accountCountryCodeKeys).toSet

}

def convertToPlaceTweetScore(

popTweetsInPlace: Seq[PopTweetsInPlace]

): Seq[PlaceTweetScore] = {

popTweetsInPlace.flatMap {

case p =>

p.popTweets.map {

case popTweet => PlaceTweetScore(p.place, popTweet.tweetId, popTweet.score)

}

}

}

def sortGeoHashTweets(

placeTweetScores: Seq[PlaceTweetScore],

rankingFunction: TopTweetsForGeoRankingFunction.Value

): Seq[PlaceTweetScore] = {

rankingFunction match {

case TopTweetsForGeoRankingFunction.Score =>

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

case TopTweetsForGeoRankingFunction.GeohashLengthAndThenScore =>

placeTweetScores

.sortBy(row => (row.place.length, row.score))(Ordering[(Int, Double)].reverse)

}

}

def getResultsForLambdaStore(

inputTarget: Target,

geohash: Option[String],

store: ReadableStore[String, PopTweetsInPlace],

topk: Int,

version: PopGeoTweetVersion.Value

): Future[Seq[(Long, Double)]] = {

inputTarget.accountCountryCode.flatMap { countryCode =>

val keys = {

if (inputTarget.params(FS.EnableCountryCodeBackoffTopTweetsByGeo))

constructKeys(geohash, countryCode, inputTarget.params(FS.GeoHashLengthList), version)

else

constructKeys(geohash, None, inputTarget.params(FS.GeoHashLengthList), version)

}

FutureOps

.mapCollect(store.multiGet(keys)).map {

case geohashTweetMap =>

val popTweets =

geohashTweetMap.values.flatten.toSeq

val results = sortGeoHashTweets(

convertToPlaceTweetScore(popTweets),

inputTarget.params(FS.RankingFunctionForTopTweetsByGeo))

.map(\_.toTweetScore).take(topk)

results

}

}

}

def getPopGeoTweetsForLoggedOutUsers(

inputTarget: Target,

store: ReadableStore[String, PopTweetsInPlace]

): Future[Seq[(Long, Double)]] = {

inputTarget.countryCode.flatMap { countryCode =>

val keys = constructKeys(None, countryCode, Seq(4), PopGeoTweetVersion.Prod)

FutureOps.mapCollect(store.multiGet(keys)).map {

case tweetMap =>

val tweets = tweetMap.values.flatten.toSeq

loggedOutPopByGeoV2CandidatesCounter.incr(tweets.size)

val popTweets = sortGeoHashTweets(

convertToPlaceTweetScore(tweets),

TopTweetsForGeoRankingFunction.Score).map(\_.toTweetScore)

popTweets

}

}

}

def getRankedTweets(

inputTarget: Target,

geohash: Option[String]

): Future[Seq[(Long, Double)]] = {

val MaxTopTweetsByGeoCandidatesToTake =

inputTarget.params(FS.MaxTopTweetsByGeoCandidatesToTake)

val scoringFn: String = inputTarget.params(FS.ScoringFuncForTopTweetsByGeo)

val combinationMethod = inputTarget.params(FS.TopTweetsByGeoCombinationParam)

val popGeoTweetVersion = inputTarget.params(FS.PopGeoTweetVersionParam)

inputTarget.isHeavyUserState.map { isHeavyUser =>

stats

.scope(combinationMethod.toString).scope(popGeoTweetVersion.toString).scope(

"IsHeavyUser\_" + isHeavyUser.toString).counter().incr()

}

combinationRequestsCounter.scope(combinationMethod.toString).counter().incr()

popGeoTweetVersionCounter.scope(popGeoTweetVersion.toString).counter().incr()

lazy val geoStoreResults = if (geohash.isDefined) {

val hash = geohash.get.take(MaxGeoHashSize)

topTweetsByGeoStore

.get(

InterestDomain[String](hash)

)

.map {

case Some(scoringFnToTweetsMapOpt) =>

val tweetsWithScore = scoringFnToTweetsMapOpt

.getOrElse(scoringFn, List.empty)

val sortedResults = sortGeoHashTweets(

tweetsWithScore.map {

case (tweetId, score) => PlaceTweetScore(hash, tweetId, score)

},

TopTweetsForGeoRankingFunction.Score

).map(\_.toTweetScore).take(

MaxTopTweetsByGeoCandidatesToTake

)

sortedResults

case \_ => Seq.empty

}

} else Future.value(Seq.empty)

lazy val versionPopGeoTweetResults =

getResultsForLambdaStore(

inputTarget,

geohash,

topTweetsByGeoStoreV2,

MaxTopTweetsByGeoCandidatesToTake,

popGeoTweetVersion

)

combinationMethod match {

case TopTweetsForGeoCombination.Default => geoStoreResults

case TopTweetsForGeoCombination.AccountsTweetFavAsBackfill =>

Future.join(geoStoreResults, versionPopGeoTweetResults).map {

case (geoStoreTweets, versionPopGeoTweets) =>

(geoStoreTweets ++ versionPopGeoTweets).take(MaxTopTweetsByGeoCandidatesToTake)

}

case TopTweetsForGeoCombination.AccountsTweetFavIntermixed =>

Future.join(geoStoreResults, versionPopGeoTweetResults).map {

case (geoStoreTweets, versionPopGeoTweets) =>

CandidateSource.interleaveSeqs(Seq(geoStoreTweets, versionPopGeoTweets))

}

}

}

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

if (inputTarget.isLoggedOutUser) {

incomingLoggedOutRequestCounter.incr()

val rankedTweets = getPopGeoTweetsForLoggedOutUsers(inputTarget, topTweetsByGeoStoreV2)

val rawCandidates = {

rankedTweets.map { rt =>

FutureOps

.mapCollect(

tweetyPieStore

.multiGet(rt.map { case (tweetId, \_) => tweetId }.toSet))

.map { tweetyPieResultMap =>

val results = buildTopTweetsByGeoRawCandidates(

inputTarget,

None,

tweetyPieResultMap

)

if (results.isEmpty) {

emptyLoggedOutRawCandidatesCounter.incr()

}

loggedOutRawCandidatesCounter.incr(results.size)

Some(results)

}

}.flatten

}

rawCandidates

} else {

incomingRequestCounter.incr()

getGeoHashForUsers(inputTarget).flatMap { geohash =>

if (geohash.isEmpty) noGeohashUserCounter.incr()

getRankedTweets(inputTarget, geohash).map { rt =>

if (rt.size == 100) {

topTweetsByGeoTake100Counter.incr(1)

}

FutureOps

.mapCollect((inputTarget.params(FS.EnableVFInTweetypie) match {

case true => tweetyPieStore

case false => tweetyPieStoreNoVF

}).multiGet(rt.map { case (tweetId, \_) => tweetId }.toSet))

.map { tweetyPieResultMap =>

Some(

buildTopTweetsByGeoRawCandidates(

inputTarget,

None,

filterOutReplyTweet(

tweetyPieResultMap,

nonReplyTweetsCounter

)

)

)

}

}.flatten

}

}

}

private def getGeoHashForUsers(

inputTarget: Target

): Future[Option[String]] = {

inputTarget.targetUser.flatMap {

case Some(user) =>

user.userType match {

case UserType.Soft =>

softUserGeoLocationStore

.get(inputTarget.targetId)

.map(\_.flatMap(\_.geohash.flatMap(\_.stringGeohash)))

case \_ =>

geoduckStoreV2.get(inputTarget.targetId).map(\_.flatMap(\_.geohash))

}

case None => Future.None

}

}

private def buildTopTweetsByGeoRawCandidates(

target: PushTypes.Target,

locationName: Option[String],

topTweets: Map[Long, Option[TweetyPieResult]]

): Seq[RawCandidate with TweetCandidate] = {

val candidates = topTweets.map { tweetIdTweetyPieResultMap =>

PushAdaptorUtil.generateOutOfNetworkTweetCandidates(

inputTarget = target,

id = tweetIdTweetyPieResultMap.\_1,

mediaCRT = MediaCRT(

CommonRecommendationType.GeoPopTweet,

CommonRecommendationType.GeoPopTweet,

CommonRecommendationType.GeoPopTweet

),

result = tweetIdTweetyPieResultMap.\_2,

localizedEntity = None

)

}.toSeq

outputTopTweetsByGeoCounter.add(candidates.length)

candidates

}

private val topTweetsByGeoFrequencyPredicate = {

TargetPredicates

.pushRecTypeFatiguePredicate(

CommonRecommendationType.GeoPopTweet,

FS.TopTweetsByGeoPushInterval,

FS.MaxTopTweetsByGeoPushGivenInterval,

stats

)

}

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

lazy val isDormantUserNotFatigued = topTweetsByGeoFrequencyPredicate(Seq(target)).map(\_.head)

lazy val enableTopTweetsByGeoForDormantUsers =

target.params(FS.EnableTopTweetsByGeoCandidatesForDormantUsers)

target.lastHTLVisitTimestamp.flatMap {

case Some(lastHTLTimestamp) =>

val minTimeSinceLastLogin =

target.params(FS.MinimumTimeSinceLastLoginForGeoPopTweetPush).ago

val timeSinceInactive = target.params(FS.TimeSinceLastLoginForGeoPopTweetPush).ago

val lastActiveTimestamp = Time.fromMilliseconds(lastHTLTimestamp)

if (lastActiveTimestamp > minTimeSinceLastLogin) {

nonDormantUsersSince14DaysCounter.incr()

Future.False

} else {

dormantUsersSince14DaysCounter.incr()

isDormantUserNotFatigued.map { isUserNotFatigued =>

lastActiveTimestamp < timeSinceInactive &&

enableTopTweetsByGeoForDormantUsers &&

isUserNotFatigued

}

}

case \_ =>

dormantUsersSince30DaysCounter.incr()

isDormantUserNotFatigued.map { isUserNotFatigued =>

enableTopTweetsByGeoForDormantUsers && isUserNotFatigued

}

}

}

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

lazy val enableTopTweetsByGeoForNewUsers = target.params(FS.EnableTopTweetsByGeoCandidates)

val isTargetEligibleForMrFatigueCheck = target.isAccountAtleastNDaysOld(

target.params(FS.MrMinDurationSincePushForTopTweetsByGeoPushes))

val isMrFatigueCheckEnabled =

target.params(FS.EnableMrMinDurationSinceMrPushFatigue)

val applyPredicateForTopTweetsByGeo =

if (isMrFatigueCheckEnabled) {

if (isTargetEligibleForMrFatigueCheck) {

DiscoverTwitterPredicate

.minDurationElapsedSinceLastMrPushPredicate(

name,

FS.MrMinDurationSincePushForTopTweetsByGeoPushes,

stats

).andThen(

topTweetsByGeoFrequencyPredicate

)(Seq(target)).map(\_.head)

} else {

Future.False

}

} else {

topTweetsByGeoFrequencyPredicate(Seq(target)).map(\_.head)

}

applyPredicateForTopTweetsByGeo.map { predicateResult =>

predicateResult && enableTopTweetsByGeoForNewUsers

}

}

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

if (target.isLoggedOutUser) {

Future.True

} else {

PushDeviceUtil

.isRecommendationsEligible(target).map(

\_ && target.params(PushParams.PopGeoCandidatesDecider)).flatMap { isAvailable =>

if (isAvailable) {

Future

.join(getAvailabilityForDormantUser(target), getAvailabilityForPlaybookSetUp(target))

.map {

case (isAvailableForDormantUser, isAvailableForPlaybook) =>

isAvailableForDormantUser || isAvailableForPlaybook

case \_ => false

}

} else Future.False

}

}

}

}