package com.twitter.frigate.pushservice.model.candidate

import com.twitter.frigate.data\_pipeline.features\_common.PushQualityModelFeatureContext.featureContext

import com.twitter.frigate.data\_pipeline.features\_common.PushQualityModelUtil

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

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

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.base.\_

import com.twitter.frigate.common.rec\_types.RecTypes

import com.twitter.frigate.common.util.NotificationScribeUtil

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

import com.twitter.frigate.pushservice.model.OutOfNetworkTweetPushCandidate

import com.twitter.frigate.pushservice.model.TopicProofTweetPushCandidate

import com.twitter.frigate.pushservice.ml.HydrationContextBuilder

import com.twitter.frigate.pushservice.predicate.quality\_model\_predicate.PDauCohort

import com.twitter.frigate.pushservice.predicate.quality\_model\_predicate.PDauCohortUtil

import com.twitter.frigate.pushservice.util.Candidate2FrigateNotification

import com.twitter.frigate.pushservice.util.MediaAnnotationsUtil.sensitiveMediaCategoryFeatureName

import com.twitter.frigate.scribe.thriftscala.FrigateNotificationScribeType

import com.twitter.frigate.scribe.thriftscala.NotificationScribe

import com.twitter.frigate.scribe.thriftscala.PredicateDetailedInfo

import com.twitter.frigate.scribe.thriftscala.PushCapInfo

import com.twitter.frigate.thriftscala.ChannelName

import com.twitter.frigate.thriftscala.FrigateNotification

import com.twitter.frigate.thriftscala.OverrideInfo

import com.twitter.gizmoduck.thriftscala.User

import com.twitter.hermit.model.user\_state.UserState.UserState

import com.twitter.ibis2.service.thriftscala.Ibis2Response

import com.twitter.ml.api.util.ScalaToJavaDataRecordConversions

import com.twitter.nrel.heavyranker.FeatureHydrator

import com.twitter.util.Future

import java.util.UUID

import java.util.concurrent.ConcurrentHashMap

import scala.collection.concurrent.{Map => CMap}

import scala.collection.Map

import scala.collection.convert.decorateAsScala.\_

trait Scriber {

self: PushCandidate =>

def statsReceiver: StatsReceiver

def frigateNotification: FrigateNotification = Candidate2FrigateNotification

.getFrigateNotification(self)(statsReceiver)

.copy(copyAggregationId = self.copyAggregationId)

lazy val impressionId: String = UUID.randomUUID.toString.replaceAll("-", "")

// Used to store the score and threshold for predicates

// Map(predicate name, (score, threshold, filter?))

private val predicateScoreAndThreshold: CMap[String, PredicateDetailedInfo] =

new ConcurrentHashMap[String, PredicateDetailedInfo]().asScala

def cachePredicateInfo(

predName: String,

predScore: Double,

predThreshold: Double,

predResult: Boolean,

additionalInformation: Option[Map[String, Double]] = None

) = {

if (!predicateScoreAndThreshold.contains(predName)) {

predicateScoreAndThreshold += predName -> PredicateDetailedInfo(

predName,

predScore,

predThreshold,

predResult,

additionalInformation)

}

}

def getCachedPredicateInfo(): Seq[PredicateDetailedInfo] = predicateScoreAndThreshold.values.toSeq

def frigateNotificationForPersistence(

channels: Seq[ChannelName],

isSilentPush: Boolean,

overrideInfoOpt: Option[OverrideInfo] = None,

copyFeaturesList: Set[String]

): Future[FrigateNotification] = {

// record display location for frigate notification

statsReceiver

.scope("FrigateNotificationForPersistence")

.scope("displayLocation")

.counter(frigateNotification.notificationDisplayLocation.name)

.incr()

val getModelScores = self.getModelScoresforScribing()

Future.join(getModelScores, self.target.targetMrUserState).map {

case (mlScores, mrUserState) =>

frigateNotification.copy(

impressionId = Some(impressionId),

isSilentPush = Some(isSilentPush),

overrideInfo = overrideInfoOpt,

mlModelScores = Some(mlScores),

mrUserState = mrUserState.map(\_.name),

copyFeatures = Some(copyFeaturesList.toSeq)

)

}

}

// scribe data

private def getNotificationScribe(

notifForPersistence: FrigateNotification,

userState: Option[UserState],

dauCohort: PDauCohort.Value,

ibis2Response: Option[Ibis2Response],

tweetAuthorId: Option[Long],

recUserId: Option[Long],

modelScoresMap: Option[Map[String, Double]],

primaryClient: Option[String],

isMrBackfillCR: Option[Boolean] = None,

tagsCR: Option[Seq[String]] = None,

gizmoduckTargetUser: Option[User],

predicateDetailedInfoList: Option[Seq[PredicateDetailedInfo]] = None,

pushCapInfoList: Option[Seq[PushCapInfo]] = None

): NotificationScribe = {

NotificationScribe(

FrigateNotificationScribeType.SendMessage,

System.currentTimeMillis(),

targetUserId = Some(self.target.targetId),

timestampKeyForHistoryV2 = Some(createdAt.inSeconds),

sendType = NotificationScribeUtil.convertToScribeDisplayLocation(

self.frigateNotification.notificationDisplayLocation

),

recommendationType = NotificationScribeUtil.convertToScribeRecommendationType(

self.frigateNotification.commonRecommendationType

),

commonRecommendationType = Some(self.frigateNotification.commonRecommendationType),

fromPushService = Some(true),

frigateNotification = Some(notifForPersistence),

impressionId = Some(impressionId),

skipModelInfo = target.skipModelInfo,

ibis2Response = ibis2Response,

tweetAuthorId = tweetAuthorId,

scribeFeatures = Some(target.noSkipButScribeFeatures),

userState = userState.map(\_.toString),

pDauCohort = Some(dauCohort.toString),

recommendedUserId = recUserId,

modelScores = modelScoresMap,

primaryClient = primaryClient,

isMrBackfillCR = isMrBackfillCR,

tagsCR = tagsCR,

targetUserType = gizmoduckTargetUser.map(\_.userType),

predicateDetailedInfoList = predicateDetailedInfoList,

pushCapInfoList = pushCapInfoList

)

}

def scribeData(

ibis2Response: Option[Ibis2Response] = None,

isSilentPush: Boolean = false,

overrideInfoOpt: Option[OverrideInfo] = None,

copyFeaturesList: Set[String] = Set.empty,

channels: Seq[ChannelName] = Seq.empty

): Future[NotificationScribe] = {

val recTweetAuthorId = self match {

case t: TweetCandidate with TweetAuthor => t.authorId

case \_ => None

}

val recUserId = self match {

case u: UserCandidate => Some(u.userId)

case \_ => None

}

val isMrBackfillCR = self match {

case t: OutOfNetworkTweetPushCandidate => t.isMrBackfillCR

case \_ => None

}

val tagsCR = self match {

case t: OutOfNetworkTweetPushCandidate =>

t.tagsCR.map { tags =>

tags.map(\_.toString)

}

case t: TopicProofTweetPushCandidate =>

t.tagsCR.map { tags =>

tags.map(\_.toString)

}

case \_ => None

}

Future

.join(

frigateNotificationForPersistence(

channels = channels,

isSilentPush = isSilentPush,

overrideInfoOpt = overrideInfoOpt,

copyFeaturesList = copyFeaturesList

),

target.targetUserState,

PDauCohortUtil.getPDauCohort(target),

target.deviceInfo,

target.targetUser

)

.flatMap {

case (notifForPersistence, userState, dauCohort, deviceInfo, gizmoduckTargetUserOpt) =>

val primaryClient = deviceInfo.flatMap(\_.guessedPrimaryClient).map(\_.toString)

val cachedPredicateInfo =

if (self.target.params(PushParams.EnablePredicateDetailedInfoScribing)) {

Some(getCachedPredicateInfo())

} else None

val cachedPushCapInfo =

if (self.target

.params(PushParams.EnablePushCapInfoScribing)) {

Some(target.finalPushcapAndFatigue.values.toSeq)

} else None

val data = getNotificationScribe(

notifForPersistence,

userState,

dauCohort,

ibis2Response,

recTweetAuthorId,

recUserId,

notifForPersistence.mlModelScores,

primaryClient,

isMrBackfillCR,

tagsCR,

gizmoduckTargetUserOpt,

cachedPredicateInfo,

cachedPushCapInfo

)

//Don't scribe features for CRTs not eligible for ML Layer

if ((target.isModelTrainingData || target.scribeFeatureWithoutHydratingNewFeatures)

&& !RecTypes.notEligibleForModelScoreTracking(self.commonRecType)) {

// scribe all the features for the model training data

self.getFeaturesForScribing.map { scribedFeatureMap =>

if (target.params(PushParams.EnableScribingMLFeaturesAsDataRecord) && !target.params(

PushFeatureSwitchParams.EnableMrScribingMLFeaturesAsFeatureMapForStaging)) {

val scribedFeatureDataRecord =

ScalaToJavaDataRecordConversions.javaDataRecord2ScalaDataRecord(

PushQualityModelUtil.adaptToDataRecord(scribedFeatureMap, featureContext))

data.copy(

featureDataRecord = Some(scribedFeatureDataRecord)

)

} else {

data.copy(features =

Some(PushQualityModelUtil.convertFeatureMapToFeatures(scribedFeatureMap)))

}

}

} else Future.value(data)

}

}

def getFeaturesForScribing: Future[FeatureMap] = {

target.featureMap

.flatMap { targetFeatureMap =>

val onlineFeatureMap = targetFeatureMap ++ self

.candidateFeatureMap() // targetFeatureMap includes target core user history features

val filteredFeatureMap = {

onlineFeatureMap.copy(

sparseContinuousFeatures = onlineFeatureMap.sparseContinuousFeatures.filterKeys(

!\_.equals(sensitiveMediaCategoryFeatureName))

)

}

val targetHydrationContext = HydrationContextBuilder.build(self.target)

val candidateHydrationContext = HydrationContextBuilder.build(self)

val featureMapFut = targetHydrationContext.join(candidateHydrationContext).flatMap {

case (targetContext, candidateContext) =>

FeatureHydrator.getFeatures(

candidateHydrationContext = candidateContext,

targetHydrationContext = targetContext,

onlineFeatures = filteredFeatureMap,

statsReceiver = statsReceiver)

}

featureMapFut

}

}

}