package com.twitter.frigate.pushservice.util

import com.twitter.finagle.stats.StatsReceiver

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

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

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

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

import com.twitter.ibis2.lib.util.JsonMarshal

import com.twitter.util.Future

import com.twitter.util.Time

object CopyUtil {

/\*\*

\* Get a list of history feature copy alone with metadata in the look back period, the metadata

\* can be used to calculate number of copy pushed after the current feature copy

\* @param candidate the candidate to be pushed to the user

\* @return Future[Seq((..,))], which is a seq of the history FEATURE copy along with

\* metadata within the look back period. In the tuple, the 4 elements represents:

\* 1. Timestamp of the past feature copy

\* 2. Option[Seq()] of copy feature names of the past copy

\* 3. Index of the particular feature copy in look back history if normal copy presents

\*/

private def getPastCopyFeaturesList(

candidate: PushCandidate

): Future[Seq[(Time, Option[Seq[String]], Int)]] = {

val target = candidate.target

target.history.map { targetHistory =>

val historyLookbackDuration = target.params(FS.CopyFeaturesHistoryLookbackDuration)

val notificationHistoryInLookbackDuration = targetHistory.sortedHistory

.takeWhile {

case (notifTimestamp, \_) => historyLookbackDuration.ago < notifTimestamp

}

notificationHistoryInLookbackDuration.zipWithIndex

.filter {

case ((\_, notification), \_) =>

notification.copyFeatures match {

case Some(copyFeatures) => copyFeatures.nonEmpty

case \_ => false

}

}

.collect {

case ((timestamp, notification), notificationIndex) =>

(timestamp, notification.copyFeatures, notificationIndex)

}

}

}

private def getPastCopyFeaturesListForF1(

candidate: PushCandidate

): Future[Seq[(Time, Option[Seq[String]], Int)]] = {

val target = candidate.target

target.history.map { targetHistory =>

val historyLookbackDuration = target.params(FS.CopyFeaturesHistoryLookbackDuration)

val notificationHistoryInLookbackDuration = targetHistory.sortedHistory

.takeWhile {

case (notifTimestamp, \_) => historyLookbackDuration.ago < notifTimestamp

}

notificationHistoryInLookbackDuration.zipWithIndex

.filter {

case ((\_, notification), \_) =>

notification.copyFeatures match {

case Some(copyFeatures) =>

RecTypes.isF1Type(notification.commonRecommendationType) && copyFeatures.nonEmpty

case \_ => false

}

}

.collect {

case ((timestamp, notification), notificationIndex) =>

(timestamp, notification.copyFeatures, notificationIndex)

}

}

}

private def getPastCopyFeaturesListForOON(

candidate: PushCandidate

): Future[Seq[(Time, Option[Seq[String]], Int)]] = {

val target = candidate.target

target.history.map { targetHistory =>

val historyLookbackDuration = target.params(FS.CopyFeaturesHistoryLookbackDuration)

val notificationHistoryInLookbackDuration = targetHistory.sortedHistory

.takeWhile {

case (notifTimestamp, \_) => historyLookbackDuration.ago < notifTimestamp

}

notificationHistoryInLookbackDuration.zipWithIndex

.filter {

case ((\_, notification), \_) =>

notification.copyFeatures match {

case Some(copyFeatures) =>

!RecTypes.isF1Type(notification.commonRecommendationType) && copyFeatures.nonEmpty

case \_ => false

}

}

.collect {

case ((timestamp, notification), notificationIndex) =>

(timestamp, notification.copyFeatures, notificationIndex)

}

}

}

private def getEmojiFeaturesMap(

candidate: PushCandidate,

copyFeatureHistory: Seq[(Time, Option[Seq[String]], Int)],

lastHTLVisitTimestamp: Option[Long],

stats: StatsReceiver

): Map[String, String] = {

val (emojiFatigueDuration, emojiFatigueNumOfPushes) = {

if (RecTypes.isF1Type(candidate.commonRecType)) {

(

candidate.target.params(FS.F1EmojiCopyFatigueDuration),

candidate.target.params(FS.F1EmojiCopyNumOfPushesFatigue))

} else {

(

candidate.target.params(FS.OonEmojiCopyFatigueDuration),

candidate.target.params(FS.OonEmojiCopyNumOfPushesFatigue))

}

}

val scopedStats = stats

.scope("getEmojiFeaturesMap").scope(candidate.commonRecType.toString).scope(

emojiFatigueDuration.toString)

val addedEmojiCopyFeature = scopedStats.counter("added\_emoji")

val fatiguedEmojiCopyFeature = scopedStats.counter("no\_emoji")

val copyFeatureType = PushConstants.EmojiFeatureNameForIbis2ModelValues

val durationFatigueCarryFunc = () =>

isUnderDurationFatigue(copyFeatureHistory, copyFeatureType, emojiFatigueDuration)

val enableHTLBasedFatigueBasicRule = candidate.target.params(FS.EnableHTLBasedFatigueBasicRule)

val minDuration = candidate.target.params(FS.MinFatigueDurationSinceLastHTLVisit)

val lastHTLVisitBasedNonFatigueWindow =

candidate.target.params(FS.LastHTLVisitBasedNonFatigueWindow)

val htlBasedCopyFatigueCarryFunc = () =>

isUnderHTLBasedFatigue(lastHTLVisitTimestamp, minDuration, lastHTLVisitBasedNonFatigueWindow)

val isUnderFatigue = getIsUnderFatigue(

Seq(

(durationFatigueCarryFunc, true),

(htlBasedCopyFatigueCarryFunc, enableHTLBasedFatigueBasicRule),

),

scopedStats

)

if (!isUnderFatigue) {

addedEmojiCopyFeature.incr()

Map(PushConstants.EmojiFeatureNameForIbis2ModelValues -> "true")

} else {

fatiguedEmojiCopyFeature.incr()

Map.empty[String, String]

}

}

private def getTargetFeaturesMap(

candidate: PushCandidate,

copyFeatureHistory: Seq[(Time, Option[Seq[String]], Int)],

lastHTLVisitTimestamp: Option[Long],

stats: StatsReceiver

): Map[String, String] = {

val targetFatigueDuration = {

if (RecTypes.isF1Type(candidate.commonRecType)) {

candidate.target.params(FS.F1TargetCopyFatigueDuration)

} else {

candidate.target.params(FS.OonTargetCopyFatigueDuration)

}

}

val scopedStats = stats

.scope("getTargetFeaturesMap").scope(candidate.commonRecType.toString).scope(

targetFatigueDuration.toString)

val addedTargetCopyFeature = scopedStats.counter("added\_target")

val fatiguedTargetCopyFeature = scopedStats.counter("no\_target")

val featureCopyType = PushConstants.TargetFeatureNameForIbis2ModelValues

val durationFatigueCarryFunc = () =>

isUnderDurationFatigue(copyFeatureHistory, featureCopyType, targetFatigueDuration)

val enableHTLBasedFatigueBasicRule = candidate.target.params(FS.EnableHTLBasedFatigueBasicRule)

val minDuration = candidate.target.params(FS.MinFatigueDurationSinceLastHTLVisit)

val lastHTLVisitBasedNonFatigueWindow =

candidate.target.params(FS.LastHTLVisitBasedNonFatigueWindow)

val htlBasedCopyFatigueCarryFunc = () =>

isUnderHTLBasedFatigue(lastHTLVisitTimestamp, minDuration, lastHTLVisitBasedNonFatigueWindow)

val isUnderFatigue = getIsUnderFatigue(

Seq(

(durationFatigueCarryFunc, true),

(htlBasedCopyFatigueCarryFunc, enableHTLBasedFatigueBasicRule),

),

scopedStats

)

if (!isUnderFatigue) {

addedTargetCopyFeature.incr()

Map(PushConstants.TargetFeatureNameForIbis2ModelValues -> "true")

} else {

fatiguedTargetCopyFeature.incr()

Map.empty[String, String]

}

}

type FatigueRuleFlag = Boolean

type FatigueRuleFunc = () => Boolean

def getIsUnderFatigue(

fatigueRulesWithFlags: Seq[(FatigueRuleFunc, FatigueRuleFlag)],

statsReceiver: StatsReceiver,

): Boolean = {

val defaultFatigue = true

val finalFatigueRes =

fatigueRulesWithFlags.zipWithIndex.foldLeft(defaultFatigue)(

(fatigueSoFar, fatigueRuleFuncWithFlagAndIndex) => {

val ((fatigueRuleFunc, flag), index) = fatigueRuleFuncWithFlagAndIndex

val funcScopedStats = statsReceiver.scope(s"fatigueFunction${index}")

if (flag) {

val shouldFatigueForTheRule = fatigueRuleFunc()

funcScopedStats.scope(s"eval\_${shouldFatigueForTheRule}").counter().incr()

val f = fatigueSoFar && shouldFatigueForTheRule

f

} else {

fatigueSoFar

}

})

statsReceiver.scope(s"final\_fatigue\_${finalFatigueRes}").counter().incr()

finalFatigueRes

}

private def isUnderDurationFatigue(

copyFeatureHistory: Seq[(Time, Option[Seq[String]], Int)],

copyFeatureType: String,

fatigueDuration: com.twitter.util.Duration,

): Boolean = {

copyFeatureHistory.exists {

case (notifTimestamp, Some(copyFeatures), \_) if copyFeatures.contains(copyFeatureType) =>

notifTimestamp > fatigueDuration.ago

case \_ => false

}

}

private def isUnderHTLBasedFatigue(

lastHTLVisitTimestamp: Option[Long],

minDurationSinceLastHTLVisit: com.twitter.util.Duration,

lastHTLVisitBasedNonFatigueWindow: com.twitter.util.Duration,

): Boolean = {

val lastHTLVisit = lastHTLVisitTimestamp.map(t => Time.fromMilliseconds(t)).getOrElse(Time.now)

val first = Time.now < (lastHTLVisit + minDurationSinceLastHTLVisit)

val second =

Time.now > (lastHTLVisit + minDurationSinceLastHTLVisit + lastHTLVisitBasedNonFatigueWindow)

first || second

}

def getOONCBasedFeature(

candidate: PushCandidate,

stats: StatsReceiver

): Future[Map[String, String]] = {

val target = candidate.target

val metric = stats.scope("getOONCBasedFeature")

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

candidate.mrWeightedOpenOrNtabClickRankingProbability.map {

case Some(score) if score >= target.params(FS.HighOONCThresholdForCopy) =>

metric.counter("high\_OONC").incr()

metric.counter(FS.HighOONCTweetFormat.toString).incr()

Map(

"whole\_template" -> JsonMarshal.toJson(

Map(

target.params(FS.HighOONCTweetFormat).toString -> true

)))

case Some(score) if score <= target.params(FS.LowOONCThresholdForCopy) =>

metric.counter("low\_OONC").incr()

metric.counter(FS.LowOONCThresholdForCopy.toString).incr()

Map(

"whole\_template" -> JsonMarshal.toJson(

Map(

target.params(FS.LowOONCTweetFormat).toString -> true

)))

case \_ =>

metric.counter("not\_in\_OONC\_range").incr()

Map.empty[String, String]

}

} else {

Future.value(Map.empty[String, String])

}

}

def getCopyFeatures(

candidate: PushCandidate,

stats: StatsReceiver,

): Future[Map[String, String]] = {

if (candidate.target.isLoggedOutUser) {

Future.value(Map.empty[String, String])

} else {

val featureMaps = getCopyBodyFeatures(candidate, stats)

for {

titleFeat <- getCopyTitleFeatures(candidate, stats)

nsfwFeat <- getNsfwCopyFeatures(candidate, stats)

ooncBasedFeature <- getOONCBasedFeature(candidate, stats)

} yield {

titleFeat ++ featureMaps ++ nsfwFeat ++ ooncBasedFeature

}

}

}

private def getCopyTitleFeatures(

candidate: PushCandidate,

stats: StatsReceiver

): Future[Map[String, String]] = {

val scopedStats = stats.scope("CopyUtil").scope("getCopyTitleFeatures")

val target = candidate.target

if ((RecTypes.isSimClusterBasedType(candidate.commonRecType) && target.params(

FS.EnableCopyFeaturesForOon)) || (RecTypes.isF1Type(candidate.commonRecType) && target

.params(FS.EnableCopyFeaturesForF1))) {

val enableTargetAndEmojiSplitFatigue = target.params(FS.EnableTargetAndEmojiSplitFatigue)

val isTargetF1Type = RecTypes.isF1Type(candidate.commonRecType)

val copyFeatureHistoryFuture = if (enableTargetAndEmojiSplitFatigue && isTargetF1Type) {

getPastCopyFeaturesListForF1(candidate)

} else if (enableTargetAndEmojiSplitFatigue && !isTargetF1Type) {

getPastCopyFeaturesListForOON(candidate)

} else {

getPastCopyFeaturesList(candidate)

}

Future

.join(

copyFeatureHistoryFuture,

target.lastHTLVisitTimestamp,

).map {

case (copyFeatureHistory, lastHTLVisitTimestamp) =>

val emojiFeatures = {

if ((RecTypes.isF1Type(candidate.commonRecType) && target.params(

FS.EnableEmojiInF1Copy))

|| RecTypes.isSimClusterBasedType(candidate.commonRecType) && target.params(

FS.EnableEmojiInOonCopy)) {

getEmojiFeaturesMap(

candidate,

copyFeatureHistory,

lastHTLVisitTimestamp,

scopedStats)

} else Map.empty[String, String]

}

val targetFeatures = {

if ((RecTypes.isF1Type(candidate.commonRecType) && target.params(

FS.EnableTargetInF1Copy)) || (RecTypes.isSimClusterBasedType(

candidate.commonRecType) && target.params(FS.EnableTargetInOonCopy))) {

getTargetFeaturesMap(

candidate,

copyFeatureHistory,

lastHTLVisitTimestamp,

scopedStats)

} else Map.empty[String, String]

}

val baseCopyFeaturesMap =

if (emojiFeatures.nonEmpty || targetFeatures.nonEmpty)

Map(PushConstants.EnableCopyFeaturesForIbis2ModelValues -> "true")

else Map.empty[String, String]

baseCopyFeaturesMap ++ emojiFeatures ++ targetFeatures

case \_ =>

Map.empty[String, String]

}

} else Future.value(Map.empty[String, String])

}

private def getCopyBodyTruncateFeatures(

candidate: PushCandidate,

): Map[String, String] = {

if (candidate.target.params(FS.EnableIosCopyBodyTruncate)) {

Map("enable\_body\_truncate\_ios" -> "true")

} else {

Map.empty[String, String]

}

}

private def getNsfwCopyFeatures(

candidate: PushCandidate,

stats: StatsReceiver

): Future[Map[String, String]] = {

val scopedStats = stats.scope("CopyUtil").scope("getNsfwCopyBodyFeatures")

val hasNsfwScoreF1Counter = scopedStats.counter("f1\_has\_nsfw\_score")

val hasNsfwScoreOonCounter = scopedStats.counter("oon\_has\_nsfw\_score")

val noNsfwScoreCounter = scopedStats.counter("no\_nsfw\_score")

val nsfwScoreF1 = scopedStats.stat("f1\_nsfw\_score")

val nsfwScoreOon = scopedStats.stat("oon\_nsfw\_score")

val isNsfwF1Counter = scopedStats.counter("is\_f1\_nsfw")

val isNsfwOonCounter = scopedStats.counter("is\_oon\_nsfw")

val target = candidate.target

val nsfwScoreFut = if (target.params(FS.EnableNsfwCopy)) {

candidate.mrNsfwScore

} else Future.None

nsfwScoreFut.map {

case Some(nsfwScore) =>

if (RecTypes.isF1Type(candidate.commonRecType)) {

hasNsfwScoreF1Counter.incr()

nsfwScoreF1.add(nsfwScore.toFloat \* 10000)

if (nsfwScore > target.params(FS.NsfwScoreThresholdForF1Copy)) {

isNsfwF1Counter.incr()

Map("is\_f1\_nsfw" -> "true")

} else {

Map.empty[String, String]

}

} else if (RecTypes.isOutOfNetworkTweetRecType(candidate.commonRecType)) {

nsfwScoreOon.add(nsfwScore.toFloat \* 10000)

hasNsfwScoreOonCounter.incr()

if (nsfwScore > target.params(FS.NsfwScoreThresholdForOONCopy)) {

isNsfwOonCounter.incr()

Map("is\_oon\_nsfw" -> "true")

} else {

Map.empty[String, String]

}

} else {

Map.empty[String, String]

}

case \_ =>

noNsfwScoreCounter.incr()

Map.empty[String, String]

}

}

private def getCopyBodyFeatures(

candidate: PushCandidate,

stats: StatsReceiver

): Map[String, String] = {

val target = candidate.target

val scopedStats = stats.scope("CopyUtil").scope("getCopyBodyFeatures")

val copyBodyFeatures = {

if (RecTypes.isF1Type(candidate.commonRecType) && target.params(FS.EnableF1CopyBody)) {

scopedStats.counter("f1BodyExpEnabled").incr()

Map(PushConstants.CopyBodyExpIbisModelValues -> "true")

} else if (RecTypes.isOutOfNetworkTweetRecType(candidate.commonRecType) && target.params(

FS.EnableOONCopyBody)) {

scopedStats.counter("oonBodyExpEnabled").incr()

Map(PushConstants.CopyBodyExpIbisModelValues -> "true")

} else

Map.empty[String, String]

}

val copyBodyTruncateFeatures = getCopyBodyTruncateFeatures(candidate)

copyBodyFeatures ++ copyBodyTruncateFeatures

}

}