package com.twitter.frigate.pushservice.predicate

import com.twitter.abuse.detection.scoring.thriftscala.TweetScoringRequest

import com.twitter.abuse.detection.scoring.thriftscala.TweetScoringResponse

import com.twitter.abuse.detection.scoring.thriftscala.{Model => TweetHealthModel}

import com.twitter.finagle.stats.Counter

import com.twitter.finagle.stats.StatsReceiver

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

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

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

import com.twitter.frigate.pushservice.params.NsfwTextDetectionModel

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

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

import com.twitter.frigate.pushservice.util.CandidateHydrationUtil

import com.twitter.frigate.pushservice.util.CandidateUtil

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

import com.twitter.frigate.thriftscala.UserMediaRepresentation

import com.twitter.hermit.predicate.NamedPredicate

import com.twitter.hermit.predicate.Predicate

import com.twitter.hss.api.thriftscala.UserHealthSignal.\_

import com.twitter.hss.api.thriftscala.SignalValue

import com.twitter.hss.api.thriftscala.UserHealthSignalResponse

import com.twitter.storehaus.ReadableStore

import com.twitter.util.Future

import com.twitter.util.Time

object HealthPredicates {

private val NsfwTextDetectionModelMap: Map[NsfwTextDetectionModel.Value, TweetHealthModel] =

Map(

NsfwTextDetectionModel.ProdModel -> TweetHealthModel.PnsfwTweetText,

NsfwTextDetectionModel.RetrainedModel -> TweetHealthModel.ExperimentalHealthModelScore1,

)

private def tweetIsSupportedLanguage(

candidate: PushCandidate,

supportedLanguages: Set[String]

): Boolean = {

val tweetLanguage =

candidate.categoricalFeatures.getOrElse("RecTweet.TweetyPieResult.Language", "")

supportedLanguages.contains(tweetLanguage)

}

def tweetHealthSignalScorePredicate(

tweetHealthScoreStore: ReadableStore[TweetScoringRequest, TweetScoringResponse],

applyToQuoteTweet: Boolean = false

)(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate with TweetCandidate with TweetDetails] = {

val name = "tweet\_health\_signal\_store\_applyToQuoteTweet\_" + applyToQuoteTweet.toString

val scopedStatsReceiver = stats.scope(name)

val numCandidatesStats = scopedStatsReceiver.scope("num\_candidates")

val numCandidatesMediaNsfwScoreStats = numCandidatesStats.scope("media\_nsfw\_score")

Predicate

.fromAsync { candidate: PushCandidate with TweetCandidate with TweetDetails =>

numCandidatesStats.counter("all").incr()

val target = candidate.target

val tweetIdOpt = if (!applyToQuoteTweet) {

Some(candidate.tweetId)

} else candidate.tweetyPieResult.flatMap(\_.quotedTweet.map(\_.id))

tweetIdOpt match {

case Some(tweetId) =>

val pMediaNsfwRequest =

TweetScoringRequest(tweetId, TweetHealthModel.ExperimentalHealthModelScore4)

tweetHealthScoreStore.get(pMediaNsfwRequest).map {

case Some(tweetScoringResponse) =>

numCandidatesMediaNsfwScoreStats.counter("non\_empty").incr()

val pMediaNsfwScore = tweetScoringResponse.score

if (!applyToQuoteTweet) {

candidate

.cacheExternalScore("NsfwMediaProbability", Future.value(Some(pMediaNsfwScore)))

}

val pMediaNsfwShouldBucket =

pMediaNsfwScore > target.params(

PushFeatureSwitchParams.PnsfwTweetMediaBucketingThreshold)

if (CandidateUtil.shouldApplyHealthQualityFilters(

candidate) && pMediaNsfwShouldBucket) {

numCandidatesMediaNsfwScoreStats.counter("bucketed").incr()

if (target.params(PushFeatureSwitchParams.PnsfwTweetMediaFilterOonOnly)

&& !RecTypes.isOutOfNetworkTweetRecType(candidate.commonRecType)) {

true

} else {

val pMediaNsfwScoreThreshold =

if (applyToQuoteTweet)

target.params(PushFeatureSwitchParams.PnsfwQuoteTweetThreshold)

else if (candidate.hasPhoto)

target.params(PushFeatureSwitchParams.PnsfwTweetImageThreshold)

else target.params(PushFeatureSwitchParams.PnsfwTweetMediaThreshold)

candidate.cachePredicateInfo(

name + "\_nsfwMedia",

pMediaNsfwScore,

pMediaNsfwScoreThreshold,

pMediaNsfwScore > pMediaNsfwScoreThreshold)

if (pMediaNsfwScore > pMediaNsfwScoreThreshold) {

numCandidatesMediaNsfwScoreStats.counter("filtered").incr()

false

} else true

}

} else true

case \_ =>

numCandidatesMediaNsfwScoreStats.counter("empty").incr()

if (candidate.hasPhoto || candidate.hasVideo) {

numCandidatesMediaNsfwScoreStats.counter("media\_tweet\_with\_empty\_score").incr()

}

true

}

case \_ => Future.True

}

}

.withStats(stats.scope(s"predicate\_$name"))

.withName(name)

}

def healthSignalScoreSpammyTweetPredicate(

tweetHealthScoreStore: ReadableStore[TweetScoringRequest, TweetScoringResponse]

)(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate with TweetCandidate with TweetDetails] = {

val name = "health\_signal\_store\_spammy\_tweet"

val statsScope = stats.scope(name)

val allCandidatesCounter = statsScope.counter("all\_candidates")

val eligibleCandidatesCounter = statsScope.counter("eligible\_candidates")

val oonCandidatesCounter = statsScope.counter("oon\_candidates")

val inCandidatesCounter = statsScope.counter("in\_candidates")

val bucketedCandidatesCounter = statsScope.counter("num\_bucketed")

val nonEmptySpamScoreCounter = statsScope.counter("non\_empty\_spam\_score")

val filteredOonCandidatesCounter = statsScope.counter("num\_filtered\_oon")

val filteredInCandidatesCounter = statsScope.counter("num\_filtered\_in")

Predicate

.fromAsync { candidate: PushCandidate with TweetCandidate with TweetDetails =>

allCandidatesCounter.incr()

val crt = candidate.commonRecType

val isOonCandidate = RecTypes.isOutOfNetworkTweetRecType(crt) ||

RecTypes.outOfNetworkTopicTweetTypes.contains(crt)

if (isOonCandidate) {

oonCandidatesCounter.incr()

}

val target = candidate.target

if (target.params(PushFeatureSwitchParams.EnableSpammyTweetFilter)) {

eligibleCandidatesCounter.incr()

val tweetSpamScore =

TweetScoringRequest(candidate.tweetId, TweetHealthModel.SpammyTweetContent)

tweetHealthScoreStore.get(tweetSpamScore).map {

case (Some(tweetScoringResponse)) =>

nonEmptySpamScoreCounter.incr()

val candidateSpamScore = tweetScoringResponse.score

candidate

.cacheExternalScore("SpammyTweetScore", Future.value(Some(candidateSpamScore)))

val tweetSpamShouldBucket =

candidateSpamScore > target.params(

PushFeatureSwitchParams.SpammyTweetBucketingThreshold)

if (CandidateUtil.shouldApplyHealthQualityFilters(

candidate) && tweetSpamShouldBucket) {

bucketedCandidatesCounter.incr()

if (isOonCandidate) {

val spamScoreThreshold =

target.params(PushFeatureSwitchParams.SpammyTweetOonThreshold)

if (candidateSpamScore > spamScoreThreshold) {

filteredOonCandidatesCounter.incr()

false

} else true

} else {

inCandidatesCounter.incr()

val spamScoreThreshold =

target.params(PushFeatureSwitchParams.SpammyTweetInThreshold)

if (candidateSpamScore > spamScoreThreshold) {

filteredInCandidatesCounter.incr()

false

} else true

}

} else true

case \_ => true

}

} else Future.True

}

.withStats(stats.scope(s"predicate\_$name"))

.withName(name)

}

def healthSignalScorePnsfwTweetTextPredicate(

tweetHealthScoreStore: ReadableStore[TweetScoringRequest, TweetScoringResponse]

)(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate with TweetCandidate] = {

val name = "health\_signal\_store\_pnsfw\_tweet\_text"

val statsScope = stats.scope(name)

val allCandidatesCounter = statsScope.counter("all\_candidates")

val nonEmptyNsfwTextScoreNum = statsScope.counter("non\_empty\_nsfw\_text\_score")

val filteredCounter = statsScope.counter("num\_filtered")

val lowScoreCounter = statsScope.counter("low\_score\_count")

Predicate

.fromAsync { candidate: PushCandidate with TweetCandidate =>

val target = candidate.target

val predEnabled =

target.params(PushFeatureSwitchParams.EnableHealthSignalStorePnsfwTweetTextPredicate)

if (CandidateUtil.shouldApplyHealthQualityFilters(

candidate) && predEnabled && tweetIsSupportedLanguage(candidate, Set(""))) {

allCandidatesCounter.incr()

val pnsfwTextRequest =

TweetScoringRequest(candidate.tweetId, TweetHealthModel.PnsfwTweetText)

tweetHealthScoreStore.get(pnsfwTextRequest).flatMap {

case Some(tweetScoringResponse) => {

nonEmptyNsfwTextScoreNum.incr()

if (tweetScoringResponse.score < 1e-8) {

lowScoreCounter.incr()

}

candidate

.cacheExternalScore(

"NsfwTextProbability-en",

Future.value(Some(tweetScoringResponse.score)))

val threshold = target.params(PushFeatureSwitchParams.PnsfwTweetTextThreshold)

candidate.cachePredicateInfo(

name,

tweetScoringResponse.score,

threshold,

tweetScoringResponse.score > threshold)

if (tweetScoringResponse.score > threshold) {

filteredCounter.incr()

Future.False

} else Future.True

}

case \_ => Future.True

}

} else Future.True

}

.withStats(stats.scope(s"predicate\_$name"))

.withName(name)

}

def healthSignalScoreMultilingualPnsfwTweetTextPredicate(

tweetHealthScoreStore: ReadableStore[TweetScoringRequest, TweetScoringResponse]

)(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate with TweetCandidate] = {

val name = "health\_signal\_store\_multilingual\_pnsfw\_tweet\_text"

val statsScope = stats.scope(name)

val allLanguagesIdentifier = "all"

val languagesSelectedForStats =

Set("") + allLanguagesIdentifier

val candidatesCounterMap: Map[String, Counter] = languagesSelectedForStats.map { lang =>

lang -> statsScope.counter(f"candidates\_$lang")

}.toMap

val nonEmptyHealthScoreMap: Map[String, Counter] = languagesSelectedForStats.map { lang =>

lang -> statsScope.counter(f"non\_empty\_health\_score\_$lang")

}.toMap

val emptyHealthScoreMap: Map[String, Counter] = languagesSelectedForStats.map { lang =>

lang -> statsScope.counter(f"empty\_health\_score\_$lang")

}.toMap

val bucketedCounterMap: Map[String, Counter] = languagesSelectedForStats.map { lang =>

lang -> statsScope.counter(f"num\_candidates\_bucketed\_$lang")

}.toMap

val filteredCounterMap: Map[String, Counter] = languagesSelectedForStats.map { lang =>

lang -> statsScope.counter(f"num\_filtered\_$lang")

}.toMap

val lowScoreCounterMap: Map[String, Counter] = languagesSelectedForStats.map { lang =>

lang -> statsScope.counter(f"low\_score\_count\_$lang")

}.toMap

val wrongBucketingModelCounter = statsScope.counter("wrong\_bucketing\_model\_count")

val wrongDetectionModelCounter = statsScope.counter("wrong\_detection\_model\_count")

def increaseCounterForLanguage(counterMap: Map[String, Counter], language: String): Unit = {

counterMap.get(allLanguagesIdentifier) match {

case Some(counter) => counter.incr()

case \_ =>

}

counterMap.get(language) match {

case Some(counter) => counter.incr()

case \_ =>

}

}

Predicate

.fromAsync { candidate: PushCandidate with TweetCandidate =>

val target = candidate.target

val languageFeatureName = "RecTweet.TweetyPieResult.Language"

lazy val isPredicateEnabledForTarget = target.params(

PushFeatureSwitchParams.EnableHealthSignalStoreMultilingualPnsfwTweetTextPredicate)

lazy val targetNsfwTextDetectionModel: NsfwTextDetectionModel.Value =

target.params(PushFeatureSwitchParams.MultilingualPnsfwTweetTextModel)

lazy val targetPredicateSupportedLanguageSeq: Seq[String] =

target.params(PushFeatureSwitchParams.MultilingualPnsfwTweetTextSupportedLanguages)

lazy val bucketingModelSeq: Seq[NsfwTextDetectionModel.Value] =

target.params(PushFeatureSwitchParams.MultilingualPnsfwTweetTextBucketingModelList)

lazy val bucketingThresholdPerLanguageSeq: Seq[Double] =

target.params(PushFeatureSwitchParams.MultilingualPnsfwTweetTextBucketingThreshold)

lazy val filteringThresholdPerLanguageSeq: Seq[Double] =

target.params(PushFeatureSwitchParams.MultilingualPnsfwTweetTextFilteringThreshold)

if (CandidateUtil.shouldApplyHealthQualityFilters(

candidate) && isPredicateEnabledForTarget) {

val candidateLanguage =

candidate.categoricalFeatures.getOrElse(languageFeatureName, "")

val indexOfCandidateLanguage =

targetPredicateSupportedLanguageSeq.indexOf(candidateLanguage)

val isCandidateLanguageSupported = indexOfCandidateLanguage >= 0

if (isCandidateLanguageSupported) {

increaseCounterForLanguage(candidatesCounterMap, candidateLanguage)

val bucketingModelScoreMap: Map[NsfwTextDetectionModel.Value, Future[Option[Double]]] =

bucketingModelSeq.map { modelName =>

NsfwTextDetectionModelMap.get(modelName) match {

case Some(targetNsfwTextDetectionModel) =>

val pnsfwTweetTextRequest: TweetScoringRequest =

TweetScoringRequest(candidate.tweetId, targetNsfwTextDetectionModel)

val scoreOptFut: Future[Option[Double]] =

tweetHealthScoreStore.get(pnsfwTweetTextRequest).map(\_.map(\_.score))

candidate

.cacheExternalScore("NsfwTextProbability", scoreOptFut)

modelName -> scoreOptFut

case \_ =>

wrongBucketingModelCounter.incr()

modelName -> Future.None

}

}.toMap

val candidateLanguageBucketingThreshold =

bucketingThresholdPerLanguageSeq(indexOfCandidateLanguage)

val userShouldBeBucketedFut: Future[Boolean] =

Future

.collect(bucketingModelScoreMap.map {

case (\_, modelScoreOptFut) =>

modelScoreOptFut.map {

case Some(score) =>

increaseCounterForLanguage(nonEmptyHealthScoreMap, candidateLanguage)

score > candidateLanguageBucketingThreshold

case \_ =>

increaseCounterForLanguage(emptyHealthScoreMap, candidateLanguage)

false

}

}.toSeq).map(\_.contains(true))

val candidateShouldBeFilteredFut: Future[Boolean] = userShouldBeBucketedFut.flatMap {

userShouldBeBucketed =>

if (userShouldBeBucketed) {

increaseCounterForLanguage(bucketedCounterMap, candidateLanguage)

val candidateLanguageFilteringThreshold =

filteringThresholdPerLanguageSeq(indexOfCandidateLanguage)

bucketingModelScoreMap.get(targetNsfwTextDetectionModel) match {

case Some(scoreOptFut) =>

scoreOptFut.map {

case Some(score) =>

val candidateShouldBeFiltered =

score > candidateLanguageFilteringThreshold

if (candidateShouldBeFiltered) {

increaseCounterForLanguage(filteredCounterMap, candidateLanguage)

}

candidateShouldBeFiltered

case \_ => false

}

case \_ =>

wrongDetectionModelCounter.incr()

Future.False

}

} else {

increaseCounterForLanguage(lowScoreCounterMap, candidateLanguage)

Future.False

}

}

candidateShouldBeFilteredFut.map(result => !result)

} else Future.True

} else Future.True

}

.withStats(stats.scope(s"predicate\_$name"))

.withName(name)

}

def authorProfileBasedPredicate(

)(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate with TweetCandidate] = {

val name = "author\_profile"

val statsScope = stats.scope(name)

val filterByNsfwToken = statsScope.counter("filter\_by\_nsfw\_token")

val filterByAccountAge = statsScope.counter("filter\_by\_account\_age")

Predicate

.fromAsync { candidate: PushCandidate with TweetCandidate =>

val target = candidate.target

candidate match {

case cand: PushCandidate with TweetAuthorDetails =>

cand.tweetAuthor.map {

case Some(author) =>

val nsfwTokens = target.params(PushFeatureSwitchParams.NsfwTokensParam)

val accountAgeInHours =

(Time.now - Time.fromMilliseconds(author.createdAtMsec)).inHours

val isNsfwAccount = CandidateHydrationUtil.isNsfwAccount(author, nsfwTokens)

val isVerified = author.safety.map(\_.verified).getOrElse(false)

if (CandidateUtil.shouldApplyHealthQualityFilters(candidate) && !isVerified) {

val enableNsfwTokenCheck =

target.params(PushFeatureSwitchParams.EnableNsfwTokenBasedFiltering)

val minimumAllowedAge =

target.params(PushFeatureSwitchParams.MinimumAllowedAuthorAccountAgeInHours)

cand.cachePredicateInfo(

name + "\_nsfwToken",

if (isNsfwAccount) 1.0 else 0.0,

0.0,

enableNsfwTokenCheck && isNsfwAccount)

cand.cachePredicateInfo(

name + "\_authorAge",

accountAgeInHours,

minimumAllowedAge,

accountAgeInHours < minimumAllowedAge)

if (enableNsfwTokenCheck && isNsfwAccount) {

filterByNsfwToken.incr()

false

} else if (accountAgeInHours < minimumAllowedAge) {

filterByAccountAge.incr()

false

} else true

} else true

case \_ => true

}

case \_ => Future.value(true)

}

}

.withStats(stats.scope(s"predicate\_$name"))

.withName(name)

}

def authorSensitiveMediaPredicate(

producerMediaRepresentationStore: ReadableStore[Long, UserMediaRepresentation]

)(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate with TweetAuthor] = {

val name = "author\_sensitive\_media\_mrtwistly"

val statsScope = stats.scope(name)

val enableQueryNum = statsScope.counter("enable\_query")

val nonEmptyMediaRepresentationNum = statsScope.counter("non\_empty\_media\_representation")

val filteredOON = statsScope.counter("filtered\_oon")

Predicate

.fromAsync { candidate: PushCandidate with TweetAuthor =>

val target = candidate.target

val useAggressiveThresholds = CandidateUtil.useAggressiveHealthThresholds(candidate)

if (CandidateUtil.shouldApplyHealthQualityFilters(candidate) &&

RecTypes.isOutOfNetworkTweetRecType(candidate.commonRecType) &&

target.params(PushFeatureSwitchParams.EnableQueryAuthorMediaRepresentationStore)) {

enableQueryNum.incr()

candidate.authorId match {

case Some(authorId) =>

producerMediaRepresentationStore.get(authorId).map {

case Some(mediaRepresentation) =>

nonEmptyMediaRepresentationNum.incr()

val sumScore: Double = mediaRepresentation.mediaRepresentation.values.sum

val nudityScore: Double = mediaRepresentation.mediaRepresentation

.getOrElse(MediaAnnotationsUtil.nudityCategoryId, 0.0)

val nudityRate = if (sumScore > 0) nudityScore / sumScore else 0.0

candidate

.cacheExternalScore("AuthorNudityScore", Future.value(Some(nudityScore)))

candidate.cacheExternalScore("AuthorNudityRate", Future.value(Some(nudityRate)))

val threshold = if (useAggressiveThresholds) {

target.params(

PushFeatureSwitchParams.AuthorSensitiveMediaFilteringThresholdForMrTwistly)

} else {

target.params(PushFeatureSwitchParams.AuthorSensitiveMediaFilteringThreshold)

}

candidate.cachePredicateInfo(

name,

nudityRate,

threshold,

nudityRate > threshold,

Some(Map[String, Double]("sumScore" -> sumScore, "nudityScore" -> nudityScore)))

if (nudityRate > threshold) {

filteredOON.incr()

false

} else true

case \_ => true

}

case \_ => Future.True

}

} else {

Future.True

}

}

.withStats(stats.scope(s"predicate\_$name"))

.withName(name)

}

def sensitiveMediaCategoryPredicate(

)(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate with TweetCandidate] = {

val name = "sensitive\_media\_category"

val tweetMediaAnnotationFeature =

"tweet.mediaunderstanding.tweet\_annotations.sensitive\_category\_probabilities"

val scopedStatsReceiver = stats.scope(name)

val allCandidatesCounter = scopedStatsReceiver.counter("all\_candidates")

val nonZeroNudityCandidatesCounter = scopedStatsReceiver.counter("non\_zero\_nudity\_candidates")

val nudityScoreStats = scopedStatsReceiver.stat("nudity\_scores")

Predicate

.fromAsync { candidate: PushCandidate =>

allCandidatesCounter.incr()

val target = candidate.target

val nudityScore = candidate.sparseContinuousFeatures

.getOrElse(tweetMediaAnnotationFeature, Map.empty[String, Double]).getOrElse(

MediaAnnotationsUtil.nudityCategoryId,

0.0)

if (nudityScore > 0) nonZeroNudityCandidatesCounter.incr()

nudityScoreStats.add(nudityScore.toFloat)

val threshold =

target.params(PushFeatureSwitchParams.TweetMediaSensitiveCategoryThresholdParam)

candidate.cachePredicateInfo(name, nudityScore, threshold, nudityScore > threshold)

if (CandidateUtil.shouldApplyHealthQualityFilters(candidate) && nudityScore > threshold) {

Future.False

} else {

Future.True

}

}

.withStats(stats.scope(s"predicate\_$name"))

.withName(name)

}

def profanityPredicate(

)(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate with TweetCandidate] = {

val name = "profanity\_filter"

val scopedStatsReceiver = stats.scope(name)

val allCandidatesCounter = scopedStatsReceiver.counter("all\_candidates")

Predicate

.fromAsync { candidate: PushCandidate =>

allCandidatesCounter.incr()

val target = candidate.target

lazy val enableFilter =

target.params(PushFeatureSwitchParams.EnableProfanityFilterParam)

val tweetSemanticCoreIds = candidate.sparseBinaryFeatures

.getOrElse(PushConstants.TweetSemanticCoreIdFeature, Set.empty[String])

if (CandidateUtil.shouldApplyHealthQualityFilters(candidate) &&

tweetSemanticCoreIds.contains(PushConstants.ProfanityFilter\_Id) && enableFilter) {

Future.False

} else {

Future.True

}

}

.withStats(stats.scope(s"predicate\_$name"))

.withName(name)

}

def agathaAbusiveTweetAuthorPredicateMrTwistly(

)(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate with OutOfNetworkTweetCandidate] = {

val name = "agatha\_abusive\_tweet\_author\_mr\_twistly"

val scopedStatsReceiver = stats.scope(name)

val allCandidatesCounter = scopedStatsReceiver.counter("all\_candidates")

val isMrBackfillCRCandidateCounter = scopedStatsReceiver.counter("isMrBackfillCR\_candidates")

Predicate

.fromAsync { cand: PushCandidate with OutOfNetworkTweetCandidate =>

allCandidatesCounter.incr()

val target = cand.target

val tweetSemanticCoreIds = cand.sparseBinaryFeatures

.getOrElse(PushConstants.TweetSemanticCoreIdFeature, Set.empty[String])

val hasAbuseStrikeTop2Percent =

tweetSemanticCoreIds.contains(PushConstants.AbuseStrike\_Top2Percent\_Id)

val hasAbuseStrikeTop1Percent =

tweetSemanticCoreIds.contains(PushConstants.AbuseStrike\_Top1Percent\_Id)

val hasAbuseStrikeTop05Percent =

tweetSemanticCoreIds.contains(PushConstants.AbuseStrike\_Top05Percent\_Id)

if (hasAbuseStrikeTop2Percent) {

scopedStatsReceiver.counter("abuse\_strike\_top\_2\_percent\_candidates").incr()

}

if (hasAbuseStrikeTop1Percent) {

scopedStatsReceiver.counter("abuse\_strike\_top\_1\_percent\_candidates").incr()

}

if (hasAbuseStrikeTop05Percent) {

scopedStatsReceiver.counter("abuse\_strike\_top\_05\_percent\_candidates").incr()

}

if (CandidateUtil.shouldApplyHealthQualityFilters(cand) && cand.isMrBackfillCR.getOrElse(

false)) {

isMrBackfillCRCandidateCounter.incr()

if (hasAbuseStrikeTop2Percent) {

if (target.params(

PushFeatureSwitchParams.EnableAbuseStrikeTop2PercentFilterSimCluster) && hasAbuseStrikeTop2Percent ||

target.params(

PushFeatureSwitchParams.EnableAbuseStrikeTop1PercentFilterSimCluster) && hasAbuseStrikeTop1Percent ||

target.params(

PushFeatureSwitchParams.EnableAbuseStrikeTop05PercentFilterSimCluster) && hasAbuseStrikeTop05Percent) {

Future.False

} else {

Future.True

}

} else {

Future.True

}

} else Future.True

}

.withStats(stats.scope(s"predicate\_$name"))

.withName(name)

}

def userHealthSignalsPredicate(

userHealthSignalStore: ReadableStore[Long, UserHealthSignalResponse]

)(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate with TweetDetails] = {

val name = "agatha\_user\_health\_model\_score"

val scopedStatsReceiver = stats.scope(name)

val allCandidatesCounter = scopedStatsReceiver.counter("all\_candidates")

val bucketedUserCandidatesCounter =

scopedStatsReceiver.counter("bucketed\_user\_candidates")

val filteredOON = scopedStatsReceiver.counter("filtered\_oon")

Predicate

.fromAsync { candidate: PushCandidate with TweetDetails =>

allCandidatesCounter.incr()

val target = candidate.target

val useAggressiveThresholds = CandidateUtil.useAggressiveHealthThresholds(candidate)

if (CandidateUtil.shouldApplyHealthQualityFilters(candidate) && target.params(

PushFeatureSwitchParams.EnableAgathaUserHealthModelPredicate)) {

val healthSignalsResponseFutOpt: Future[Option[UserHealthSignalResponse]] =

candidate.authorId match {

case Some(authorId) => userHealthSignalStore.get(authorId)

case \_ => Future.None

}

healthSignalsResponseFutOpt.map {

case Some(response) =>

val agathaRecentAbuseStrikeScore: Double = userHealthSignalValueToDouble(

response.signalValues

.getOrElse(AgathaRecentAbuseStrikeDouble, SignalValue.DoubleValue(0.0)))

val agathaCalibratedNSFWScore: Double = userHealthSignalValueToDouble(

response.signalValues

.getOrElse(AgathaCalibratedNsfwDouble, SignalValue.DoubleValue(0.0)))

val agathaTextNSFWScore: Double = userHealthSignalValueToDouble(response.signalValues

.getOrElse(NsfwTextUserScoreDouble, SignalValue.DoubleValue(0.0)))

candidate

.cacheExternalScore(

"agathaRecentAbuseStrikeScore",

Future.value(Some(agathaRecentAbuseStrikeScore)))

candidate

.cacheExternalScore(

"agathaCalibratedNSFWScore",

Future.value(Some(agathaCalibratedNSFWScore)))

candidate

.cacheExternalScore("agathaTextNSFWScore", Future.value(Some(agathaTextNSFWScore)))

val NSFWShouldBucket = agathaCalibratedNSFWScore > target.params(

PushFeatureSwitchParams.AgathaCalibratedNSFWBucketThreshold)

val textNSFWShouldBucket = agathaTextNSFWScore > target.params(

PushFeatureSwitchParams.AgathaTextNSFWBucketThreshold)

if (NSFWShouldBucket || textNSFWShouldBucket) {

bucketedUserCandidatesCounter.incr()

if (NSFWShouldBucket) {

scopedStatsReceiver.counter("calibrated\_nsfw\_bucketed\_user\_candidates").incr()

}

if (textNSFWShouldBucket) {

scopedStatsReceiver.counter("text\_nsfw\_bucketed\_user\_candidates").incr()

}

val (thresholdAgathaNsfw, thresholdTextNsfw) = if (useAggressiveThresholds) {

(

target.params(

PushFeatureSwitchParams.AgathaCalibratedNSFWThresholdForMrTwistly),

target

.params(PushFeatureSwitchParams.AgathaTextNSFWThresholdForMrTwistly))

} else {

(

target.params(PushFeatureSwitchParams.AgathaCalibratedNSFWThreshold),

target.params(PushFeatureSwitchParams.AgathaTextNSFWThreshold))

}

candidate.cachePredicateInfo(

name + "\_agathaNsfw",

agathaCalibratedNSFWScore,

thresholdAgathaNsfw,

agathaCalibratedNSFWScore > thresholdAgathaNsfw)

candidate.cachePredicateInfo(

name + "\_authorTextNsfw",

agathaTextNSFWScore,

thresholdTextNsfw,

agathaTextNSFWScore > thresholdTextNsfw)

if ((agathaCalibratedNSFWScore > thresholdAgathaNsfw) ||

(agathaTextNSFWScore > thresholdTextNsfw)) {

filteredOON.incr()

false

} else true

} else {

true

}

case \_ => true

}

} else {

Future.True

}

}

.withStats(stats.scope(s"predicate\_$name"))

.withName(name)

}

def userHealthSignalValueToDouble(signalValue: SignalValue): Double = {

signalValue match {

case SignalValue.DoubleValue(value) => value

case \_ => throw new Exception(f"Could not convert signal value to double")

}

}

}