package com.twitter.frigate.pushservice.predicate.ntab\_caret\_fatigue

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.notificationservice.thriftscala.GenericType

import com.twitter.hermit.predicate.NamedPredicate

import com.twitter.notificationservice.genericfeedbackstore.FeedbackPromptValue

import com.twitter.hermit.predicate.Predicate

import com.twitter.frigate.common.base.Candidate

import com.twitter.frigate.common.base.RecommendationType

import com.twitter.frigate.common.base.TargetInfo

import com.twitter.frigate.thriftscala.CommonRecommendationType

import com.twitter.frigate.thriftscala.SeeLessOftenType

import com.twitter.frigate.common.history.History

import com.twitter.frigate.common.predicate.FrigateHistoryFatiguePredicate.TimeSeries

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

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

import com.twitter.frigate.common.predicate.ntab\_caret\_fatigue.NtabCaretClickFatiguePredicateHelper

import com.twitter.frigate.pushservice.predicate.CaretFeedbackHistoryFilter

import com.twitter.notificationservice.thriftscala.CaretFeedbackDetails

import com.twitter.util.Duration

import com.twitter.util.Future

import com.twitter.frigate.common.predicate.FatiguePredicate

import com.twitter.frigate.pushservice.util.PushCapUtil

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

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

object CRTBasedNtabCaretClickFatiguePredicates {

private val MagicRecsCategory = "MagicRecs"

private val HighQualityRefreshableTypes: Set[Option[String]] = Set(

Some("MagicRecHighQualityTweet"),

)

private def getUserStateWeight(target: Target): Future[Double] = {

PushDeviceUtil.isNtabOnlyEligible.map {

case true =>

target.params(PushFeatureSwitchParams.SeeLessOftenNtabOnlyNotifUserPushCapWeight)

case \_ => 1.0

}

}

def crtToSeeLessOftenType(

crt: CommonRecommendationType,

candidate: Candidate

with RecommendationType

with TargetInfo[

Target

],

): SeeLessOftenType = {

val crtToSeeLessOftenTypeMap: Map[CommonRecommendationType, SeeLessOftenType] = {

RecTypes.f1FirstDegreeTypes.map((\_, SeeLessOftenType.F1Type)).toMap

}

crtToSeeLessOftenTypeMap.getOrElse(crt, SeeLessOftenType.OtherTypes)

}

def genericTypeToSeeLessOftenType(

genericType: GenericType,

candidate: Candidate

with RecommendationType

with TargetInfo[

Target

]

): SeeLessOftenType = {

val genericTypeToSeeLessOftenTypeMap: Map[GenericType, SeeLessOftenType] = {

Map(GenericType.MagicRecFirstDegreeTweetRecent -> SeeLessOftenType.F1Type)

}

genericTypeToSeeLessOftenTypeMap.getOrElse(genericType, SeeLessOftenType.OtherTypes)

}

def getWeightForCaretFeedback(

dislikedType: SeeLessOftenType,

candidate: Candidate

with RecommendationType

with TargetInfo[

Target

]

): Double = {

def getWeightFromDislikedAndCurrentType(

dislikedType: SeeLessOftenType,

currentType: SeeLessOftenType

): Double = {

val weightMap: Map[(SeeLessOftenType, SeeLessOftenType), Double] = {

Map(

(SeeLessOftenType.F1Type, SeeLessOftenType.F1Type) -> candidate.target.params(

PushFeatureSwitchParams.SeeLessOftenF1TriggerF1PushCapWeight),

(SeeLessOftenType.OtherTypes, SeeLessOftenType.OtherTypes) -> candidate.target.params(

PushFeatureSwitchParams.SeeLessOftenNonF1TriggerNonF1PushCapWeight),

(SeeLessOftenType.F1Type, SeeLessOftenType.OtherTypes) -> candidate.target.params(

PushFeatureSwitchParams.SeeLessOftenF1TriggerNonF1PushCapWeight),

(SeeLessOftenType.OtherTypes, SeeLessOftenType.F1Type) -> candidate.target.params(

PushFeatureSwitchParams.SeeLessOftenNonF1TriggerF1PushCapWeight)

)

}

weightMap

.getOrElse(

(dislikedType, currentType),

candidate.target.params(PushFeatureSwitchParams.SeeLessOftenDefaultPushCapWeight))

}

getWeightFromDislikedAndCurrentType(

dislikedType,

crtToSeeLessOftenType(candidate.commonRecType, candidate))

}

private def isOutsideCrtBasedNtabCaretClickFatiguePeriodContFn(

candidate: Candidate

with RecommendationType

with TargetInfo[

Target

],

history: History,

feedbackDetails: Seq[CaretFeedbackDetails],

filterHistory: TimeSeries => TimeSeries =

FatiguePredicate.recTypesOnlyFilter(RecTypes.sharedNTabCaretFatigueTypes),

filterCaretFeedbackHistory: Target => Seq[

CaretFeedbackDetails

] => Seq[CaretFeedbackDetails] =

CaretFeedbackHistoryFilter.caretFeedbackHistoryFilter(Seq(MagicRecsCategory)),

knobs: Seq[Double],

pushCapKnobs: Seq[Double],

powerKnobs: Seq[Double],

f1Weight: Double,

nonF1Weight: Double,

defaultPushCap: Int,

stats: StatsReceiver,

tripHqTweetWeight: Double = 0.0,

): Boolean = {

val filteredFeedbackDetails = filterCaretFeedbackHistory(candidate.target)(feedbackDetails)

val weight = {

if (RecTypes.HighQualityTweetTypes.contains(

candidate.commonRecType) && (tripHqTweetWeight != 0)) {

tripHqTweetWeight

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

f1Weight

} else {

nonF1Weight

}

}

val filteredHistory = History(filterHistory(history.history.toSeq).toMap)

isOutsideFatiguePeriod(

filteredHistory,

filteredFeedbackDetails,

Seq(),

ContinuousFunctionParam(

knobs,

pushCapKnobs,

powerKnobs,

weight,

defaultPushCap

),

stats.scope(

if (RecTypes.isF1Type(candidate.commonRecType)) "mr\_ntab\_dislike\_f1\_candidate\_fn"

else if (RecTypes.HighQualityTweetTypes.contains(candidate.commonRecType))

"mr\_ntab\_dislike\_high\_quality\_candidate\_fn"

else "mr\_ntab\_dislike\_nonf1\_candidate\_fn")

)

}

private def isOutsideFatiguePeriod(

history: History,

feedbackDetails: Seq[CaretFeedbackDetails],

feedbacks: Seq[FeedbackModel],

param: ContinuousFunctionParam,

stats: StatsReceiver

): Boolean = {

val fatiguePeriod: Duration =

NtabCaretClickFatigueUtils.durationToFilterForFeedback(

feedbackDetails,

feedbacks,

param,

param.defaultValue,

stats

)

val hasRecentSent =

NtabCaretClickFatiguePredicateHelper.hasRecentSend(history, fatiguePeriod)

!hasRecentSent

}

def genericCRTBasedNtabCaretClickFnFatiguePredicate[

Cand <: Candidate with RecommendationType with TargetInfo[

Target

]

](

filterHistory: TimeSeries => TimeSeries =

FatiguePredicate.recTypesOnlyFilter(RecTypes.sharedNTabCaretFatigueTypes),

filterCaretFeedbackHistory: Target => Seq[

CaretFeedbackDetails

] => Seq[CaretFeedbackDetails] = CaretFeedbackHistoryFilter

.caretFeedbackHistoryFilter(Seq(MagicRecsCategory)),

filterInlineFeedbackHistory: Seq[FeedbackModel] => Seq[FeedbackModel] =

NtabCaretClickFatigueUtils.feedbackModelFilterByCRT(RecTypes.sharedNTabCaretFatigueTypes)

)(

implicit stats: StatsReceiver

): NamedPredicate[Cand] = {

val predicateName = "generic\_crt\_based\_ntab\_dislike\_fatigue\_fn"

Predicate

.fromAsync[Cand] { cand: Cand =>

{

if (!cand.target.params(PushFeatureSwitchParams.EnableGenericCRTBasedFatiguePredicate)) {

Future.True

} else {

val scopedStats = stats.scope(predicateName)

val totalRequests = scopedStats.counter("mr\_ntab\_dislike\_total")

val total90Day =

scopedStats.counter("mr\_ntab\_dislike\_90day\_dislike")

val totalDisabled =

scopedStats.counter("mr\_ntab\_dislike\_not\_90day\_dislike")

val totalSuccess = scopedStats.counter("mr\_ntab\_dislike\_success")

val totalFiltered = scopedStats.counter("mr\_ntab\_dislike\_filtered")

val totalWithHistory =

scopedStats.counter("mr\_ntab\_dislike\_with\_history")

val totalWithoutHistory =

scopedStats.counter("mr\_ntab\_dislike\_without\_history")

totalRequests.incr()

Future

.join(

cand.target.history,

cand.target.caretFeedbacks,

cand.target.dynamicPushcap,

cand.target.optoutAdjustedPushcap,

PushCapUtil.getDefaultPushCap(cand.target),

getUserStateWeight(cand.target)

).map {

case (

history,

Some(feedbackDetails),

dynamicPushcapOpt,

optoutAdjustedPushcapOpt,

defaultPushCap,

userStateWeight) => {

totalWithHistory.incr()

val feedbackDetailsDeduped =

NtabCaretClickFatiguePredicateHelper.dedupFeedbackDetails(

filterCaretFeedbackHistory(cand.target)(feedbackDetails),

stats

)

val pushCap: Int = (dynamicPushcapOpt, optoutAdjustedPushcapOpt) match {

case (\_, Some(optoutAdjustedPushcap)) => optoutAdjustedPushcap

case (Some(pushcapInfo), \_) => pushcapInfo.pushcap

case \_ => defaultPushCap

}

val filteredHistory = History(filterHistory(history.history.toSeq).toMap)

val hasUserDislikeInLast90Days =

NtabCaretClickFatigueUtils.hasUserDislikeInLast90Days(feedbackDetailsDeduped)

val isF1TriggerFatigueEnabled = cand.target

.params(PushFeatureSwitchParams.EnableContFnF1TriggerSeeLessOftenFatigue)

val isNonF1TriggerFatigueEnabled = cand.target.params(

PushFeatureSwitchParams.EnableContFnNonF1TriggerSeeLessOftenFatigue)

val isOutisdeSeeLessOftenFatigue =

if (hasUserDislikeInLast90Days && (isF1TriggerFatigueEnabled || isNonF1TriggerFatigueEnabled)) {

total90Day.incr()

val feedbackDetailsGroupedBySeeLessOftenType: Map[Option[

SeeLessOftenType

], Seq[

CaretFeedbackDetails

]] = feedbackDetails.groupBy(feedbackDetail =>

feedbackDetail.genericNotificationMetadata.map(x =>

genericTypeToSeeLessOftenType(x.genericType, cand)))

val isOutsideFatiguePeriodSeq =

for (elem <- feedbackDetailsGroupedBySeeLessOftenType if elem.\_1.isDefined)

yield {

val dislikedSeeLessOftenType: SeeLessOftenType = elem.\_1.get

val seqCaretFeedbackDetails: Seq[CaretFeedbackDetails] = elem.\_2

val weight = getWeightForCaretFeedback(

dislikedSeeLessOftenType,

cand) \* userStateWeight

if (isOutsideFatiguePeriod(

history = filteredHistory,

feedbackDetails = seqCaretFeedbackDetails,

feedbacks = Seq(),

param = ContinuousFunctionParam(

knobs = cand.target

.params(PushFeatureSwitchParams.SeeLessOftenListOfDayKnobs),

knobValues = cand.target

.params(

PushFeatureSwitchParams.SeeLessOftenListOfPushCapWeightKnobs).map(

\_ \* pushCap),

powers = cand.target

.params(PushFeatureSwitchParams.SeeLessOftenListOfPowerKnobs),

weight = weight,

defaultValue = pushCap

),

scopedStats

)) {

true

} else {

false

}

}

isOutsideFatiguePeriodSeq.forall(identity)

} else {

totalDisabled.incr()

true

}

if (isOutisdeSeeLessOftenFatigue) {

totalSuccess.incr()

} else totalFiltered.incr()

isOutisdeSeeLessOftenFatigue

}

case \_ =>

totalSuccess.incr()

totalWithoutHistory.incr()

true

}

}

}

}.withStats(stats.scope(predicateName))

.withName(predicateName)

}

def f1TriggeredCRTBasedNtabCaretClickFnFatiguePredicate[

Cand <: Candidate with RecommendationType with TargetInfo[

Target

]

](

filterHistory: TimeSeries => TimeSeries =

FatiguePredicate.recTypesOnlyFilter(RecTypes.sharedNTabCaretFatigueTypes),

filterCaretFeedbackHistory: Target => Seq[

CaretFeedbackDetails

] => Seq[CaretFeedbackDetails] = CaretFeedbackHistoryFilter

.caretFeedbackHistoryFilter(Seq(MagicRecsCategory)),

filterInlineFeedbackHistory: Seq[FeedbackModel] => Seq[FeedbackModel] =

NtabCaretClickFatigueUtils.feedbackModelFilterByCRT(RecTypes.sharedNTabCaretFatigueTypes)

)(

implicit stats: StatsReceiver

): NamedPredicate[Cand] = {

val predicateName = "f1\_triggered\_crt\_based\_ntab\_dislike\_fatigue\_fn"

Predicate

.fromAsync[Cand] { cand: Cand =>

{

val scopedStats = stats.scope(predicateName)

val totalRequests = scopedStats.counter("mr\_ntab\_dislike\_total")

val total90Day =

scopedStats.counter("mr\_ntab\_dislike\_90day\_dislike")

val totalDisabled =

scopedStats.counter("mr\_ntab\_dislike\_not\_90day\_dislike")

val totalSuccess = scopedStats.counter("mr\_ntab\_dislike\_success")

val totalFiltered = scopedStats.counter("mr\_ntab\_dislike\_filtered")

val totalWithHistory =

scopedStats.counter("mr\_ntab\_dislike\_with\_history")

val totalWithoutHistory =

scopedStats.counter("mr\_ntab\_dislike\_without\_history")

totalRequests.incr()

Future

.join(

cand.target.history,

cand.target.caretFeedbacks,

cand.target.dynamicPushcap,

cand.target.optoutAdjustedPushcap,

cand.target.notificationFeedbacks,

PushCapUtil.getDefaultPushCap(cand.target),

getUserStateWeight(cand.target)

).map {

case (

history,

Some(feedbackDetails),

dynamicPushcapOpt,

optoutAdjustedPushcapOpt,

Some(feedbacks),

defaultPushCap,

userStateWeight) =>

totalWithHistory.incr()

val feedbackDetailsDeduped =

NtabCaretClickFatiguePredicateHelper.dedupFeedbackDetails(

filterCaretFeedbackHistory(cand.target)(feedbackDetails),

stats

)

val pushCap: Int = (dynamicPushcapOpt, optoutAdjustedPushcapOpt) match {

case (\_, Some(optoutAdjustedPushcap)) => optoutAdjustedPushcap

case (Some(pushcapInfo), \_) => pushcapInfo.pushcap

case \_ => defaultPushCap

}

val filteredHistory = History(filterHistory(history.history.toSeq).toMap)

val isOutsideInlineDislikeFatigue =

if (cand.target

.params(PushFeatureSwitchParams.EnableContFnF1TriggerInlineFeedbackFatigue)) {

val weight =

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

cand.target

.params(PushFeatureSwitchParams.InlineFeedbackF1TriggerF1PushCapWeight)

} else {

cand.target

.params(PushFeatureSwitchParams.InlineFeedbackF1TriggerNonF1PushCapWeight)

}

val inlineFeedbackFatigueParam = ContinuousFunctionParam(

cand.target

.params(PushFeatureSwitchParams.InlineFeedbackListOfDayKnobs),

cand.target

.params(PushFeatureSwitchParams.InlineFeedbackListOfPushCapWeightKnobs)

.map(\_ \* pushCap),

cand.target

.params(PushFeatureSwitchParams.InlineFeedbackListOfPowerKnobs),

weight,

pushCap

)

isInlineDislikeOutsideFatiguePeriod(

cand,

feedbacks

.collect {

case feedbackPromptValue: FeedbackPromptValue =>

InlineFeedbackModel(feedbackPromptValue, None)

},

filteredHistory,

Seq(

filterInlineFeedbackHistory,

NtabCaretClickFatigueUtils.feedbackModelFilterByCRT(

RecTypes.f1FirstDegreeTypes)),

inlineFeedbackFatigueParam,

scopedStats

)

} else true

lazy val isOutsidePromptDislikeFatigue =

if (cand.target

.params(PushFeatureSwitchParams.EnableContFnF1TriggerPromptFeedbackFatigue)) {

val weight =

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

cand.target

.params(PushFeatureSwitchParams.PromptFeedbackF1TriggerF1PushCapWeight)

} else {

cand.target

.params(PushFeatureSwitchParams.PromptFeedbackF1TriggerNonF1PushCapWeight)

}

val promptFeedbackFatigueParam = ContinuousFunctionParam(

cand.target

.params(PushFeatureSwitchParams.PromptFeedbackListOfDayKnobs),

cand.target

.params(PushFeatureSwitchParams.PromptFeedbackListOfPushCapWeightKnobs)

.map(\_ \* pushCap),

cand.target

.params(PushFeatureSwitchParams.PromptFeedbackListOfPowerKnobs),

weight,

pushCap

)

isPromptDislikeOutsideFatiguePeriod(

feedbacks

.collect {

case feedbackPromptValue: FeedbackPromptValue =>

PromptFeedbackModel(feedbackPromptValue, None)

},

filteredHistory,

Seq(

filterInlineFeedbackHistory,

NtabCaretClickFatigueUtils.feedbackModelFilterByCRT(

RecTypes.f1FirstDegreeTypes)),

promptFeedbackFatigueParam,

scopedStats

)

} else true

isOutsideInlineDislikeFatigue && isOutsidePromptDislikeFatigue

case \_ =>

totalSuccess.incr()

totalWithoutHistory.incr()

true

}

}

}.withStats(stats.scope(predicateName))

.withName(predicateName)

}

def nonF1TriggeredCRTBasedNtabCaretClickFnFatiguePredicate[

Cand <: Candidate with RecommendationType with TargetInfo[

Target

]

](

filterHistory: TimeSeries => TimeSeries =

FatiguePredicate.recTypesOnlyFilter(RecTypes.sharedNTabCaretFatigueTypes),

filterCaretFeedbackHistory: Target => Seq[

CaretFeedbackDetails

] => Seq[CaretFeedbackDetails] = CaretFeedbackHistoryFilter

.caretFeedbackHistoryFilter(Seq(MagicRecsCategory)),

filterInlineFeedbackHistory: Seq[FeedbackModel] => Seq[FeedbackModel] =

NtabCaretClickFatigueUtils.feedbackModelFilterByCRT(RecTypes.sharedNTabCaretFatigueTypes)

)(

implicit stats: StatsReceiver

): NamedPredicate[Cand] = {

val predicateName = "non\_f1\_triggered\_crt\_based\_ntab\_dislike\_fatigue\_fn"

Predicate

.fromAsync[Cand] { cand: Cand =>

{

val scopedStats = stats.scope(predicateName)

val totalRequests = scopedStats.counter("mr\_ntab\_dislike\_total")

val total90Day =

scopedStats.counter("mr\_ntab\_dislike\_90day\_dislike")

val totalDisabled =

scopedStats.counter("mr\_ntab\_dislike\_not\_90day\_dislike")

val totalSuccess = scopedStats.counter("mr\_ntab\_dislike\_success")

val totalFiltered = scopedStats.counter("mr\_ntab\_dislike\_filtered")

val totalWithHistory =

scopedStats.counter("mr\_ntab\_dislike\_with\_history")

val totalWithoutHistory =

scopedStats.counter("mr\_ntab\_dislike\_without\_history")

val totalFeedbackSuccess = scopedStats.counter("mr\_total\_feedback\_success")

totalRequests.incr()

Future

.join(

cand.target.history,

cand.target.caretFeedbacks,

cand.target.dynamicPushcap,

cand.target.optoutAdjustedPushcap,

cand.target.notificationFeedbacks,

PushCapUtil.getDefaultPushCap(cand.target),

getUserStateWeight(cand.target),

).map {

case (

history,

Some(feedbackDetails),

dynamicPushcapOpt,

optoutAdjustedPushcapOpt,

Some(feedbacks),

defaultPushCap,

userStateWeight) =>

totalWithHistory.incr()

val filteredfeedbackDetails =

if (cand.target.params(

PushFeatureSwitchParams.AdjustTripHqTweetTriggeredNtabCaretClickFatigue)) {

val refreshableTypeFilter = CaretFeedbackHistoryFilter

.caretFeedbackHistoryFilterByRefreshableTypeDenyList(

HighQualityRefreshableTypes)

refreshableTypeFilter(cand.target)(feedbackDetails)

} else {

feedbackDetails

}

val feedbackDetailsDeduped =

NtabCaretClickFatiguePredicateHelper.dedupFeedbackDetails(

filterCaretFeedbackHistory(cand.target)(filteredfeedbackDetails),

stats

)

val pushCap: Int = (dynamicPushcapOpt, optoutAdjustedPushcapOpt) match {

case (\_, Some(optoutAdjustedPushcap)) => optoutAdjustedPushcap

case (Some(pushcapInfo), \_) => pushcapInfo.pushcap

case \_ => defaultPushCap

}

val filteredHistory = History(filterHistory(history.history.toSeq).toMap)

val isOutsideInlineDislikeFatigue =

if (cand.target

.params(

PushFeatureSwitchParams.EnableContFnNonF1TriggerInlineFeedbackFatigue)) {

val weight =

if (RecTypes.isF1Type(cand.commonRecType))

cand.target

.params(PushFeatureSwitchParams.InlineFeedbackNonF1TriggerF1PushCapWeight)

else

cand.target

.params(

PushFeatureSwitchParams.InlineFeedbackNonF1TriggerNonF1PushCapWeight)

val inlineFeedbackFatigueParam = ContinuousFunctionParam(

cand.target

.params(PushFeatureSwitchParams.InlineFeedbackListOfDayKnobs),

cand.target

.params(PushFeatureSwitchParams.InlineFeedbackListOfPushCapWeightKnobs)

.map(\_ \* pushCap),

cand.target

.params(PushFeatureSwitchParams.InlineFeedbackListOfPowerKnobs),

weight,

pushCap

)

val excludedCRTs: Set[CommonRecommendationType] =

if (cand.target.params(

PushFeatureSwitchParams.AdjustTripHqTweetTriggeredNtabCaretClickFatigue)) {

RecTypes.f1FirstDegreeTypes ++ RecTypes.HighQualityTweetTypes

} else {

RecTypes.f1FirstDegreeTypes

}

isInlineDislikeOutsideFatiguePeriod(

cand,

feedbacks

.collect {

case feedbackPromptValue: FeedbackPromptValue =>

InlineFeedbackModel(feedbackPromptValue, None)

},

filteredHistory,

Seq(

filterInlineFeedbackHistory,

NtabCaretClickFatigueUtils.feedbackModelExcludeCRT(excludedCRTs)),

inlineFeedbackFatigueParam,

scopedStats

)

} else true

lazy val isOutsidePromptDislikeFatigue =

if (cand.target

.params(

PushFeatureSwitchParams.EnableContFnNonF1TriggerPromptFeedbackFatigue)) {

val weight =

if (RecTypes.isF1Type(cand.commonRecType))

cand.target

.params(PushFeatureSwitchParams.PromptFeedbackNonF1TriggerF1PushCapWeight)

else

cand.target

.params(

PushFeatureSwitchParams.PromptFeedbackNonF1TriggerNonF1PushCapWeight)

val promptFeedbackFatigueParam = ContinuousFunctionParam(

cand.target

.params(PushFeatureSwitchParams.PromptFeedbackListOfDayKnobs),

cand.target

.params(PushFeatureSwitchParams.PromptFeedbackListOfPushCapWeightKnobs)

.map(\_ \* pushCap),

cand.target

.params(PushFeatureSwitchParams.PromptFeedbackListOfPowerKnobs),

weight,

pushCap

)

isPromptDislikeOutsideFatiguePeriod(

feedbacks

.collect {

case feedbackPromptValue: FeedbackPromptValue =>

PromptFeedbackModel(feedbackPromptValue, None)

},

filteredHistory,

Seq(

filterInlineFeedbackHistory,

NtabCaretClickFatigueUtils.feedbackModelExcludeCRT(

RecTypes.f1FirstDegreeTypes)),

promptFeedbackFatigueParam,

scopedStats

)

} else true

isOutsideInlineDislikeFatigue && isOutsidePromptDislikeFatigue

case \_ =>

totalFeedbackSuccess.incr()

totalWithoutHistory.incr()

true

}

}

}.withStats(stats.scope(predicateName))

.withName(predicateName)

}

def tripHqTweetTriggeredCRTBasedNtabCaretClickFnFatiguePredicate[

Cand <: Candidate with RecommendationType with TargetInfo[

Target

]

](

filterHistory: TimeSeries => TimeSeries =

FatiguePredicate.recTypesOnlyFilter(RecTypes.sharedNTabCaretFatigueTypes),

filterCaretFeedbackHistory: Target => Seq[

CaretFeedbackDetails

] => Seq[CaretFeedbackDetails] = CaretFeedbackHistoryFilter

.caretFeedbackHistoryFilter(Seq(MagicRecsCategory)),

filterInlineFeedbackHistory: Seq[FeedbackModel] => Seq[FeedbackModel] =

NtabCaretClickFatigueUtils.feedbackModelFilterByCRT(RecTypes.sharedNTabCaretFatigueTypes)

)(

implicit stats: StatsReceiver

): NamedPredicate[Cand] = {

val predicateName = "trip\_hq\_tweet\_triggered\_crt\_based\_ntab\_dislike\_fatigue\_fn"

Predicate

.fromAsync[Cand] { cand: Cand =>

{

val scopedStats = stats.scope(predicateName)

val totalRequests = scopedStats.counter("mr\_ntab\_dislike\_total")

val total90Day =

scopedStats.counter("mr\_ntab\_dislike\_90day\_dislike")

val totalDisabled =

scopedStats.counter("mr\_ntab\_dislike\_not\_90day\_dislike")

val totalSuccess = scopedStats.counter("mr\_ntab\_dislike\_success")

val totalFiltered = scopedStats.counter("mr\_ntab\_dislike\_filtered")

val totalWithHistory =

scopedStats.counter("mr\_ntab\_dislike\_with\_history")

val totalWithoutHistory =

scopedStats.counter("mr\_ntab\_dislike\_without\_history")

val totalFeedbackSuccess = scopedStats.counter("mr\_total\_feedback\_success")

totalRequests.incr()

Future

.join(

cand.target.history,

cand.target.caretFeedbacks,

cand.target.dynamicPushcap,

cand.target.optoutAdjustedPushcap,

cand.target.notificationFeedbacks,

PushCapUtil.getDefaultPushCap(cand.target),

getUserStateWeight(cand.target),

).map {

case (

history,

Some(feedbackDetails),

dynamicPushcapOpt,

optoutAdjustedPushcapOpt,

Some(feedbacks),

defaultPushCap,

userStateWeight) =>

totalWithHistory.incr()

if (cand.target.params(

PushFeatureSwitchParams.AdjustTripHqTweetTriggeredNtabCaretClickFatigue)) {

val refreshableTypeFilter = CaretFeedbackHistoryFilter

.caretFeedbackHistoryFilterByRefreshableType(HighQualityRefreshableTypes)

val filteredfeedbackDetails = refreshableTypeFilter(cand.target)(feedbackDetails)

val feedbackDetailsDeduped =

NtabCaretClickFatiguePredicateHelper.dedupFeedbackDetails(

filterCaretFeedbackHistory(cand.target)(filteredfeedbackDetails),

stats

)

val pushCap: Int = (dynamicPushcapOpt, optoutAdjustedPushcapOpt) match {

case (\_, Some(optoutAdjustedPushcap)) => optoutAdjustedPushcap

case (Some(pushcapInfo), \_) => pushcapInfo.pushcap

case \_ => defaultPushCap

}

val filteredHistory = History(filterHistory(history.history.toSeq).toMap)

val isOutsideInlineDislikeFatigue =

if (cand.target

.params(

PushFeatureSwitchParams.EnableContFnNonF1TriggerInlineFeedbackFatigue)) {

val weight = {

if (RecTypes.HighQualityTweetTypes.contains(cand.commonRecType)) {

cand.target

.params(

PushFeatureSwitchParams.InlineFeedbackNonF1TriggerNonF1PushCapWeight)

} else {

cand.target

.params(

PushFeatureSwitchParams.InlineFeedbackNonF1TriggerF1PushCapWeight)

}

}

val inlineFeedbackFatigueParam = ContinuousFunctionParam(

cand.target

.params(PushFeatureSwitchParams.InlineFeedbackListOfDayKnobs),

cand.target

.params(PushFeatureSwitchParams.InlineFeedbackListOfPushCapWeightKnobs)

.map(\_ \* pushCap),

cand.target

.params(PushFeatureSwitchParams.InlineFeedbackListOfPowerKnobs),

weight,

pushCap

)

val includedCRTs: Set[CommonRecommendationType] =

RecTypes.HighQualityTweetTypes

isInlineDislikeOutsideFatiguePeriod(

cand,

feedbacks

.collect {

case feedbackPromptValue: FeedbackPromptValue =>

InlineFeedbackModel(feedbackPromptValue, None)

},

filteredHistory,

Seq(

filterInlineFeedbackHistory,

NtabCaretClickFatigueUtils.feedbackModelFilterByCRT(includedCRTs)),

inlineFeedbackFatigueParam,

scopedStats

)

} else true

lazy val isOutsidePromptDislikeFatigue =

if (cand.target

.params(

PushFeatureSwitchParams.EnableContFnNonF1TriggerPromptFeedbackFatigue)) {

val weight =

if (RecTypes.isF1Type(cand.commonRecType))

cand.target

.params(

PushFeatureSwitchParams.PromptFeedbackNonF1TriggerF1PushCapWeight)

else

cand.target

.params(

PushFeatureSwitchParams.PromptFeedbackNonF1TriggerNonF1PushCapWeight)

val promptFeedbackFatigueParam = ContinuousFunctionParam(

cand.target

.params(PushFeatureSwitchParams.PromptFeedbackListOfDayKnobs),

cand.target

.params(PushFeatureSwitchParams.PromptFeedbackListOfPushCapWeightKnobs)

.map(\_ \* pushCap),

cand.target

.params(PushFeatureSwitchParams.PromptFeedbackListOfPowerKnobs),

weight,

pushCap

)

isPromptDislikeOutsideFatiguePeriod(

feedbacks

.collect {

case feedbackPromptValue: FeedbackPromptValue =>

PromptFeedbackModel(feedbackPromptValue, None)

},

filteredHistory,

Seq(

filterInlineFeedbackHistory,

NtabCaretClickFatigueUtils.feedbackModelExcludeCRT(

RecTypes.f1FirstDegreeTypes)),

promptFeedbackFatigueParam,

scopedStats

)

} else true

isOutsideInlineDislikeFatigue && isOutsidePromptDislikeFatigue

} else {

true

}

case \_ =>

totalFeedbackSuccess.incr()

totalWithoutHistory.incr()

true

}

}

}.withStats(stats.scope(predicateName))

.withName(predicateName)

}

private def getDedupedInlineFeedbackByType(

inlineFeedbacks: Seq[FeedbackModel],

feedbackType: FeedbackTypeEnum.Value,

revertedFeedbackType: FeedbackTypeEnum.Value

): Seq[FeedbackModel] = {

inlineFeedbacks

.filter(feedback =>

feedback.feedbackTypeEnum == feedbackType ||

feedback.feedbackTypeEnum == revertedFeedbackType)

.groupBy(feedback => feedback.notificationImpressionId.getOrElse(""))

.toSeq

.collect {

case (impressionId, feedbacks: Seq[FeedbackModel]) if (feedbacks.nonEmpty) =>

val latestFeedback = feedbacks.maxBy(feedback => feedback.timestampMs)

if (latestFeedback.feedbackTypeEnum == feedbackType)

Some(latestFeedback)

else None

case \_ => None

}

.flatten

}

private def getDedupedInlineFeedback(

inlineFeedbacks: Seq[FeedbackModel],

target: Target

): Seq[FeedbackModel] = {

val inlineDislikeFeedback =

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

getDedupedInlineFeedbackByType(

inlineFeedbacks,

FeedbackTypeEnum.InlineDislike,

FeedbackTypeEnum.InlineRevertedDislike)

} else Seq()

val inlineDismissFeedback =

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

getDedupedInlineFeedbackByType(

inlineFeedbacks,

FeedbackTypeEnum.InlineDismiss,

FeedbackTypeEnum.InlineRevertedDismiss)

} else Seq()

val inlineSeeLessFeedback =

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

getDedupedInlineFeedbackByType(

inlineFeedbacks,

FeedbackTypeEnum.InlineSeeLess,

FeedbackTypeEnum.InlineRevertedSeeLess)

} else Seq()

val inlineNotRelevantFeedback =

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

getDedupedInlineFeedbackByType(

inlineFeedbacks,

FeedbackTypeEnum.InlineNotRelevant,

FeedbackTypeEnum.InlineRevertedNotRelevant)

} else Seq()

inlineDislikeFeedback ++ inlineDismissFeedback ++ inlineSeeLessFeedback ++ inlineNotRelevantFeedback

}

private def isInlineDislikeOutsideFatiguePeriod(

candidate: Candidate

with RecommendationType

with TargetInfo[

Target

],

inlineFeedbacks: Seq[FeedbackModel],

filteredHistory: History,

feedbackFilters: Seq[Seq[FeedbackModel] => Seq[FeedbackModel]],

inlineFeedbackFatigueParam: ContinuousFunctionParam,

stats: StatsReceiver

): Boolean = {

val scopedStats = stats.scope("inline\_dislike\_fatigue")

val inlineNegativeFeedback =

getDedupedInlineFeedback(inlineFeedbacks, candidate.target)

val hydratedInlineNegativeFeedback = FeedbackModelHydrator.HydrateNotification(

inlineNegativeFeedback,

filteredHistory.history.toSeq.map(\_.\_2))

if (isOutsideFatiguePeriod(

filteredHistory,

Seq(),

feedbackFilters.foldLeft(hydratedInlineNegativeFeedback)((feedbacks, feedbackFilter) =>

feedbackFilter(feedbacks)),

inlineFeedbackFatigueParam,

scopedStats

)) {

scopedStats.counter("feedback\_inline\_dislike\_success").incr()

true

} else {

scopedStats.counter("feedback\_inline\_dislike\_filtered").incr()

false

}

}

private def isPromptDislikeOutsideFatiguePeriod(

feedbacks: Seq[FeedbackModel],

filteredHistory: History,

feedbackFilters: Seq[Seq[FeedbackModel] => Seq[FeedbackModel]],

inlineFeedbackFatigueParam: ContinuousFunctionParam,

stats: StatsReceiver

): Boolean = {

val scopedStats = stats.scope("prompt\_dislike\_fatigue")

val promptDislikeFeedback = feedbacks

.filter(feedback => feedback.feedbackTypeEnum == FeedbackTypeEnum.PromptIrrelevant)

val hydratedPromptDislikeFeedback = FeedbackModelHydrator.HydrateNotification(

promptDislikeFeedback,

filteredHistory.history.toSeq.map(\_.\_2))

if (isOutsideFatiguePeriod(

filteredHistory,

Seq(),

feedbackFilters.foldLeft(hydratedPromptDislikeFeedback)((feedbacks, feedbackFilter) =>

feedbackFilter(feedbacks)),

inlineFeedbackFatigueParam,

scopedStats

)) {

scopedStats.counter("feedback\_prompt\_dislike\_success").incr()

true

} else {

scopedStats.counter("feedback\_prompt\_dislike\_filtered").incr()

false

}

}

}