package com.twitter.visibility.rules

import com.twitter.contenthealth.sensitivemediasettings.thriftscala.SensitiveMediaSettingsLevel

import com.twitter.contenthealth.toxicreplyfilter.thriftscala.FilterState

import com.twitter.conversions.DurationOps.\_

import com.twitter.gizmoduck.thriftscala.Label

import com.twitter.gizmoduck.thriftscala.MuteSurface

import com.twitter.health.platform\_manipulation.stcm\_tweet\_holdback.StcmTweetHoldback

import com.twitter.search.common.constants.thriftscala.ThriftQuerySource

import com.twitter.snowflake.id.SnowflakeId

import com.twitter.takedown.util.TakedownReasons

import com.twitter.takedown.util.{TakedownReasons => TakedownReasonsUtil}

import com.twitter.timelines.configapi.EnumParam

import com.twitter.timelines.configapi.Param

import com.twitter.timelines.configapi.Params

import com.twitter.tseng.withholding.thriftscala.TakedownReason

import com.twitter.util.Duration

import com.twitter.util.Time

import com.twitter.visibility.configapi.params.RuleParam

import com.twitter.visibility.configapi.params.RuleParams

import com.twitter.visibility.features.AuthorIsSuspended

import com.twitter.visibility.features.CardIsPoll

import com.twitter.visibility.features.CardUriHost

import com.twitter.visibility.features.SearchQuerySource

import com.twitter.visibility.features.\_

import com.twitter.visibility.features.{AuthorBlocksOuterAuthor => AuthorBlocksOuterAuthorFeature}

import com.twitter.visibility.features.{AuthorBlocksViewer => AuthorBlocksViewerFeature}

import com.twitter.visibility.features.{

CommunityTweetAuthorIsRemoved => CommunityTweetAuthorIsRemovedFeature

}

import com.twitter.visibility.features.{

CommunityTweetCommunityNotFound => CommunityTweetCommunityNotFoundFeature

}

import com.twitter.visibility.features.{

CommunityTweetCommunityDeleted => CommunityTweetCommunityDeletedFeature

}

import com.twitter.visibility.features.{

CommunityTweetCommunitySuspended => CommunityTweetCommunitySuspendedFeature

}

import com.twitter.visibility.features.{

CommunityTweetCommunityVisible => CommunityTweetCommunityVisibleFeature

}

import com.twitter.visibility.features.{CommunityTweetIsHidden => CommunityTweetIsHiddenFeature}

import com.twitter.visibility.features.{

NotificationIsOnCommunityTweet => NotificationIsOnCommunityTweetFeature

}

import com.twitter.visibility.features.{OuterAuthorFollowsAuthor => OuterAuthorFollowsAuthorFeature}

import com.twitter.visibility.features.{OuterAuthorIsInnerAuthor => OuterAuthorIsInnerAuthorFeature}

import com.twitter.visibility.features.{TweetHasCard => TweetHasCardFeature}

import com.twitter.visibility.features.{TweetHasMedia => TweetHasMediaFeature}

import com.twitter.visibility.features.{TweetIsCommunityTweet => TweetIsCommunityTweetFeature}

import com.twitter.visibility.features.{TweetIsEditTweet => TweetIsEditTweetFeature}

import com.twitter.visibility.features.{TweetIsStaleTweet => TweetIsStaleTweetFeature}

import com.twitter.visibility.features.{ViewerBlocksAuthor => ViewerBlocksAuthorFeature}

import com.twitter.visibility.features.{ViewerIsCommunityAdmin => ViewerIsCommunityAdminFeature}

import com.twitter.visibility.features.{ViewerIsCommunityMember => ViewerIsCommunityMemberFeature}

import com.twitter.visibility.features.{

ViewerIsCommunityModerator => ViewerIsCommunityModeratorFeature

}

import com.twitter.visibility.features.{

ViewerIsInternalCommunitiesAdmin => ViewerIsInternalCommunitiesAdminFeature

}

import com.twitter.visibility.features.{ViewerMutesAuthor => ViewerMutesAuthorFeature}

import com.twitter.visibility.features.{

ViewerMutesRetweetsFromAuthor => ViewerMutesRetweetsFromAuthorFeature

}

import com.twitter.visibility.models.ViolationLevel

import com.twitter.visibility.models.\_

import com.twitter.visibility.rules.Result.FoundCardUriRootDomain

import com.twitter.visibility.rules.Result.FoundMediaLabel

import com.twitter.visibility.rules.Result.FoundSpaceLabel

import com.twitter.visibility.rules.Result.FoundSpaceLabelWithScoreAboveThreshold

import com.twitter.visibility.rules.Result.FoundTweetLabel

import com.twitter.visibility.rules.Result.FoundTweetLabelForPerspectivalUser

import com.twitter.visibility.rules.Result.FoundTweetLabelWithLanguageIn

import com.twitter.visibility.rules.Result.FoundTweetLabelWithLanguageScoreAboveThreshold

import com.twitter.visibility.rules.Result.FoundTweetLabelWithScoreAboveThreshold

import com.twitter.visibility.rules.Result.FoundTweetViolationOfLevel

import com.twitter.visibility.rules.Result.FoundTweetViolationOfSomeLevel

import com.twitter.visibility.rules.Result.FoundUserLabel

import com.twitter.visibility.rules.Result.FoundUserRole

import com.twitter.visibility.rules.Result.HasQuerySource

import com.twitter.visibility.rules.Result.HasTweetTimestampAfterCutoff

import com.twitter.visibility.rules.Result.HasTweetTimestampAfterOffset

import com.twitter.visibility.rules.Result.HasTweetTimestampBeforeCutoff

import com.twitter.visibility.rules.Result.ParamWasTrue

import com.twitter.visibility.rules.Result.Result

import com.twitter.visibility.rules.Result.Satisfied

import com.twitter.visibility.rules.Result.Unsatisfied

import com.twitter.visibility.util.NamingUtils

import com.twitter.visibility.{features => feats}

sealed trait PreFilterResult

case object Filtered extends PreFilterResult

case object NeedsFullEvaluation extends PreFilterResult

case object NotFiltered extends PreFilterResult

sealed trait Condition {

lazy val name: String = NamingUtils.getFriendlyName(this)

def features: Set[Feature[\_]]

def optionalFeatures: Set[Feature[\_]]

def preFilter(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): PreFilterResult = {

if (features.forall(featureMap.contains)) {

if (apply(evaluationContext, featureMap).asBoolean) {

NotFiltered

} else {

Filtered

}

} else {

NeedsFullEvaluation

}

}

def apply(evaluationContext: EvaluationContext, featureMap: Map[Feature[\_], \_]): Result

}

trait PreFilterOnOptionalFeatures extends Condition {

override def preFilter(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): PreFilterResult =

if ((features ++ optionalFeatures).forall(featureMap.contains)) {

if (apply(evaluationContext, featureMap).asBoolean) {

NotFiltered

} else {

Filtered

}

} else {

NeedsFullEvaluation

}

}

trait HasSafetyLabelType {

val labelTypes: Set[SafetyLabelType]

def hasLabelType(labelType: SafetyLabelType): Boolean = labelTypes.contains(labelType)

}

sealed trait HasNestedConditions extends HasSafetyLabelType {

val conditions: Seq[Condition]

override lazy val labelTypes: Set[SafetyLabelType] = conditions

.collect {

case lt: HasSafetyLabelType => lt.labelTypes

}.flatten.toSet

}

object Result {

sealed trait ConditionReason

case object FoundInnerQuotedTweet extends ConditionReason

case object FoundTweetViolationOfSomeLevel extends ConditionReason

case class FoundTweetViolationOfLevel(level: ViolationLevel) extends ConditionReason

case class FoundTweetLabel(label: TweetSafetyLabelType) extends ConditionReason

case class FoundSpaceLabel(label: SpaceSafetyLabelType) extends ConditionReason

case class FoundMediaLabel(label: MediaSafetyLabelType) extends ConditionReason

case class FoundTweetLabelForPerspectivalUser(label: TweetSafetyLabelType) extends ConditionReason

case class FoundTweetLabelWithLanguageScoreAboveThreshold(

label: TweetSafetyLabelType,

languagesToScoreThresholds: Map[String, Double])

extends ConditionReason

case class FoundTweetLabelWithScoreAboveThreshold(label: TweetSafetyLabelType, threshold: Double)

extends ConditionReason

case class FoundTweetLabelWithLanguageIn(

safetyLabelType: TweetSafetyLabelType,

languages: Set[String])

extends ConditionReason

case class FoundTweetSafetyLabelWithPredicate(safetyLabelType: TweetSafetyLabelType, name: String)

extends ConditionReason

case class FoundUserLabel(label: UserLabelValue) extends ConditionReason

case class FoundMutedKeyword(keyword: String) extends ConditionReason

case object HasTweetTimestampAfterCutoff extends ConditionReason

case object HasTweetTimestampAfterOffset extends ConditionReason

case object HasTweetTimestampBeforeCutoff extends ConditionReason

case class IsTweetReplyToParentTweetBeforeDuration(duration: Duration) extends ConditionReason

case class IsTweetReplyToRootTweetBeforeDuration(duration: Duration) extends ConditionReason

case class HasQuerySource(querySource: ThriftQuerySource) extends ConditionReason

case class FoundUserRole(role: String) extends ConditionReason

case class ViewerInHrcj(jurisdiction: String) extends ConditionReason

case class ViewerOrRequestInCountry(country: String) extends ConditionReason

case class ViewerAgeInYears(ageInYears: Int) extends ConditionReason

case object NoViewerAge extends ConditionReason

case class ParamWasTrue(param: Param[Boolean]) extends ConditionReason

case class FoundCardUriRootDomain(domain: String) extends ConditionReason

case object Unknown extends ConditionReason

sealed trait Result {

def asBoolean: Boolean

}

val SatisfiedResult: Result = Satisfied()

case class Satisfied(reason: ConditionReason = Unknown) extends Result {

override val asBoolean: Boolean = true

}

case class Unsatisfied(condition: Condition) extends Result {

override val asBoolean: Boolean = false

}

def fromMutedKeyword(mutedKeyword: MutedKeyword, unsatisfied: Unsatisfied): Result = {

mutedKeyword match {

case MutedKeyword(Some(keyword)) => Satisfied(FoundMutedKeyword(keyword))

case \_ => unsatisfied

}

}

case class FoundSpaceLabelWithScoreAboveThreshold(label: SpaceSafetyLabelType, threshold: Double)

extends ConditionReason

}

object Condition {

abstract class BooleanFeatureCondition(feature: Feature[Boolean]) extends Condition {

override val features: Set[Feature[\_]] = Set(feature)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result =

if (featureMap(feature).asInstanceOf[Boolean]) {

Result.SatisfiedResult

} else {

UnsatisfiedResult

}

}

case class ParamIsTrue(param: Param[Boolean]) extends Condition with HasParams {

override lazy val name: String = s"ParamIsTrue(${NamingUtils.getFriendlyName(param)})"

override val features: Set[Feature[\_]] = Set.empty

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult = Unsatisfied(this)

private val SatisfiedResult = Satisfied(ParamWasTrue(param))

override val params: Set[Param[\_]] = Set(param)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result =

if (evaluationContext.params(param)) {

SatisfiedResult

} else {

UnsatisfiedResult

}

}

case object Never extends Condition {

override lazy val name: String = s"""Never"""

override val features: Set[Feature[\_]] = Set.empty

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult = Unsatisfied(this)

override def preFilter(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): PreFilterResult = {

NeedsFullEvaluation

}

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result =

UnsatisfiedResult

}

class BooleanCondition(value: Boolean) extends Condition {

override lazy val name: String = s"""${if (value) "True" else "False"}"""

override val features: Set[Feature[\_]] = Set.empty

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result =

value match {

case true => Result.SatisfiedResult

case false => UnsatisfiedResult

}

}

case object True extends BooleanCondition(true)

case object False extends BooleanCondition(false)

abstract class ContentTakendownInViewerCountry(takedownFeature: Feature[Seq[TakedownReason]])

extends Condition {

override val features: Set[Feature[\_]] = Set(takedownFeature)

override val optionalFeatures: Set[Feature[\_]] = Set(RequestCountryCode)

private val UnsatisfiedResult = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val requestCountryCode = featureMap.get(RequestCountryCode).asInstanceOf[Option[String]]

val takedownReasons = featureMap(takedownFeature).asInstanceOf[Seq[TakedownReason]]

if (TakedownReasonsUtil.isTakenDownIn(requestCountryCode, takedownReasons)) {

Result.SatisfiedResult

} else {

UnsatisfiedResult

}

}

}

case object TweetTakendownInViewerCountry

extends ContentTakendownInViewerCountry(TweetTakedownReasons)

case object AuthorTakendownInViewerCountry

extends ContentTakendownInViewerCountry(AuthorTakedownReasons)

case object SuspendedAuthor extends BooleanFeatureCondition(AuthorIsSuspended)

case object SuspendedViewer extends BooleanFeatureCondition(ViewerIsSuspended)

case object DeactivatedViewer extends BooleanFeatureCondition(ViewerIsDeactivated)

case object UnavailableAuthor extends BooleanFeatureCondition(AuthorIsUnavailable)

case object IsVerifiedCrawlerViewer extends BooleanFeatureCondition(RequestIsVerifiedCrawler)

case object LoggedOutViewer extends Condition {

override val features: Set[Feature[\_]] = Set.empty

override val optionalFeatures: Set[Feature[\_]] = Set(ViewerId)

private val UnsatisfiedResult = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result =

if (featureMap.contains(ViewerId)) UnsatisfiedResult else Result.SatisfiedResult

}

case object IsSelfQuote extends Condition {

override val features: Set[Feature[\_]] = Set(AuthorId, OuterAuthorId)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val authorIds = featureMap(AuthorId).asInstanceOf[Set[Long]]

val outerAuthorId = featureMap(OuterAuthorId).asInstanceOf[Long]

if (authorIds.contains(outerAuthorId)) {

Result.SatisfiedResult

} else {

UnsatisfiedResult

}

}

}

case object ViewerIsAuthor extends Condition {

override val features: Set[Feature[\_]] = Set(AuthorId)

override val optionalFeatures: Set[Feature[\_]] = Set(ViewerId)

private val UnsatisfiedResult = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result =

if (featureMap.contains(ViewerId)) {

val authorIds = featureMap(AuthorId).asInstanceOf[Set[Long]]

val viewerId = featureMap(ViewerId).asInstanceOf[Long]

if (authorIds.contains(viewerId)) {

Result.SatisfiedResult

} else {

UnsatisfiedResult

}

} else {

UnsatisfiedResult

}

}

case object NonAuthorViewer extends Condition {

override val features: Set[Feature[\_]] = Set(AuthorId)

override val optionalFeatures: Set[Feature[\_]] = Set(ViewerId)

private val UnsatisfiedResult = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result =

if (featureMap.contains(ViewerId)) {

val authorIds = featureMap(AuthorId).asInstanceOf[Set[Long]]

val viewerId = featureMap(ViewerId).asInstanceOf[Long]

if (authorIds.contains(viewerId)) {

UnsatisfiedResult

} else {

Result.SatisfiedResult

}

} else {

Result.SatisfiedResult

}

}

case object ViewerFollowsAuthorOfFosnrViolatingTweet

extends BooleanFeatureCondition(ViewerFollowsAuthorOfViolatingTweet)

case object ViewerDoesNotFollowAuthorOfFosnrViolatingTweet

extends BooleanFeatureCondition(ViewerDoesNotFollowAuthorOfViolatingTweet)

case object ViewerDoesFollowAuthor extends BooleanFeatureCondition(ViewerFollowsAuthor)

case object AuthorDoesFollowViewer extends BooleanFeatureCondition(AuthorFollowsViewer)

case object AuthorBlocksViewer extends BooleanFeatureCondition(AuthorBlocksViewerFeature)

case object ViewerBlocksAuthor extends BooleanFeatureCondition(ViewerBlocksAuthorFeature)

case object ViewerIsUnmentioned extends BooleanFeatureCondition(NotificationIsOnUnmentionedViewer)

case object AuthorBlocksOuterAuthor

extends BooleanFeatureCondition(AuthorBlocksOuterAuthorFeature)

case object OuterAuthorFollowsAuthor

extends BooleanFeatureCondition(OuterAuthorFollowsAuthorFeature)

case object OuterAuthorIsInnerAuthor

extends BooleanFeatureCondition(OuterAuthorIsInnerAuthorFeature)

case object ViewerMutesAuthor extends BooleanFeatureCondition(ViewerMutesAuthorFeature)

case object ViewerReportsAuthor extends BooleanFeatureCondition(ViewerReportsAuthorAsSpam)

case object ViewerReportsTweet extends BooleanFeatureCondition(ViewerReportedTweet)

case object IsQuotedInnerTweet extends BooleanFeatureCondition(TweetIsInnerQuotedTweet)

case object IsSourceTweet extends BooleanFeatureCondition(TweetIsSourceTweet)

case object ViewerMutesRetweetsFromAuthor

extends BooleanFeatureCondition(ViewerMutesRetweetsFromAuthorFeature)

case object ConversationRootAuthorDoesFollowViewer

extends BooleanFeatureCondition(ConversationRootAuthorFollowsViewer)

case object ViewerDoesFollowConversationRootAuthor

extends BooleanFeatureCondition(ViewerFollowsConversationRootAuthor)

case object TweetIsCommunityTweet extends BooleanFeatureCondition(TweetIsCommunityTweetFeature)

case object NotificationIsOnCommunityTweet

extends BooleanFeatureCondition(NotificationIsOnCommunityTweetFeature)

sealed trait CommunityTweetCommunityUnavailable extends Condition

case object CommunityTweetCommunityNotFound

extends BooleanFeatureCondition(CommunityTweetCommunityNotFoundFeature)

with CommunityTweetCommunityUnavailable

case object CommunityTweetCommunityDeleted

extends BooleanFeatureCondition(CommunityTweetCommunityDeletedFeature)

with CommunityTweetCommunityUnavailable

case object CommunityTweetCommunitySuspended

extends BooleanFeatureCondition(CommunityTweetCommunitySuspendedFeature)

with CommunityTweetCommunityUnavailable

case object CommunityTweetCommunityVisible

extends BooleanFeatureCondition(CommunityTweetCommunityVisibleFeature)

case object ViewerIsInternalCommunitiesAdmin

extends BooleanFeatureCondition(ViewerIsInternalCommunitiesAdminFeature)

case object ViewerIsCommunityAdmin extends BooleanFeatureCondition(ViewerIsCommunityAdminFeature)

case object ViewerIsCommunityModerator

extends BooleanFeatureCondition(ViewerIsCommunityModeratorFeature)

case object ViewerIsCommunityMember

extends BooleanFeatureCondition(ViewerIsCommunityMemberFeature)

sealed trait CommunityTweetIsModerated extends Condition

case object CommunityTweetIsHidden

extends BooleanFeatureCondition(CommunityTweetIsHiddenFeature)

with CommunityTweetIsModerated

case object CommunityTweetAuthorIsRemoved

extends BooleanFeatureCondition(CommunityTweetAuthorIsRemovedFeature)

with CommunityTweetIsModerated

case object DoesHaveInnerCircleOfFriendsRelationship

extends BooleanFeatureCondition(HasInnerCircleOfFriendsRelationship)

case object TweetIsCommunityConversation

extends BooleanFeatureCondition(TweetHasCommunityConversationControl)

case object TweetIsByInvitationConversation

extends BooleanFeatureCondition(TweetHasByInvitationConversationControl)

case object TweetIsFollowersConversation

extends BooleanFeatureCondition(TweetHasFollowersConversationControl)

case object ViewerIsTweetConversationRootAuthor

extends BooleanFeatureCondition(TweetConversationViewerIsRootAuthor)

private case object ViewerIsInvitedToTweetConversationByMention

extends BooleanFeatureCondition(TweetConversationViewerIsInvited)

private case object ViewerIsInvitedToTweetConversationByReplyMention

extends BooleanFeatureCondition(TweetConversationViewerIsInvitedViaReplyMention)

object ViewerIsInvitedToTweetConversation

extends Or(

ViewerIsInvitedToTweetConversationByMention,

ViewerIsInvitedToTweetConversationByReplyMention)

object TweetIsExclusiveContent extends BooleanFeatureCondition(TweetIsExclusiveTweet)

object ViewerIsExclusiveTweetAuthor

extends BooleanFeatureCondition(ViewerIsExclusiveTweetRootAuthor)

object ViewerSuperFollowsExclusiveTweetAuthor

extends BooleanFeatureCondition(ViewerSuperFollowsExclusiveTweetRootAuthor)

object TweetIsTrustedFriendsContent extends BooleanFeatureCondition(TweetIsTrustedFriendTweet)

object ViewerIsTrustedFriendsTweetAuthor

extends BooleanFeatureCondition(ViewerIsTrustedFriendTweetAuthor)

object ViewerIsTrustedFriend extends BooleanFeatureCondition(ViewerIsTrustedFriendOfTweetAuthor)

object TweetIsCollabInvitationContent

extends BooleanFeatureCondition(TweetIsCollabInvitationTweet)

case class TweetHasLabelForPerspectivalUser(safetyLabel: TweetSafetyLabelType)

extends Condition

with HasSafetyLabelType {

override lazy val name: String = s"TweetHasLabelForPerspectivalUser(${safetyLabel.name})"

override val features: Set[Feature[\_]] = Set(TweetSafetyLabels)

override val optionalFeatures: Set[Feature[\_]] = Set(ViewerId)

override val labelTypes: Set[SafetyLabelType] = Set(safetyLabel)

private val UnsatisfiedResult: Unsatisfied = Unsatisfied(this)

private val SatisfiedResult: Satisfied = Satisfied(

FoundTweetLabelForPerspectivalUser(safetyLabel)

)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

if (!featureMap.contains(ViewerId)) {

UnsatisfiedResult

} else {

val viewerId = featureMap(ViewerId).asInstanceOf[Long]

val labels = featureMap(TweetSafetyLabels).asInstanceOf[Seq[TweetSafetyLabel]]

labels

.collectFirst {

case label

if label.labelType == safetyLabel && label.applicableUsers.contains(viewerId)

&& ExperimentBase.shouldFilterForSource(evaluationContext.params, label.source) =>

SatisfiedResult

}.getOrElse(UnsatisfiedResult)

}

}

}

case class TweetHasLabel(

safetyLabel: TweetSafetyLabelType,

labelSourceExperimentPredicate: Option[(Params, Option[LabelSource]) => Boolean] = None)

extends Condition

with HasSafetyLabelType {

override lazy val name: String = s"TweetHasLabel(${safetyLabel.name})"

override val features: Set[Feature[\_]] = Set(TweetSafetyLabels)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

override val labelTypes: Set[SafetyLabelType] = Set(safetyLabel)

private val UnsatisfiedResult: Unsatisfied = Unsatisfied(this)

private val SatisfiedResult: Satisfied = Satisfied(FoundTweetLabel(safetyLabel))

private val labelSourcePredicate: (Params, Option[LabelSource]) => Boolean =

labelSourceExperimentPredicate match {

case Some(predicate) => predicate

case \_ => ExperimentBase.shouldFilterForSource

}

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val labels = featureMap(TweetSafetyLabels).asInstanceOf[Seq[TweetSafetyLabel]]

labels

.collectFirst {

case label

if label.labelType == safetyLabel

&& labelSourcePredicate(evaluationContext.params, label.source) =>

SatisfiedResult

}.getOrElse(UnsatisfiedResult)

}

}

case class SpaceHasLabel(

safetyLabelType: SpaceSafetyLabelType)

extends Condition

with HasSafetyLabelType {

override lazy val name: String = s"SpaceHasLabel(${safetyLabelType.name})"

override val features: Set[Feature[\_]] = Set(SpaceSafetyLabels)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

override val labelTypes: Set[SafetyLabelType] = Set(safetyLabelType)

private val UnsatisfiedResult: Unsatisfied = Unsatisfied(this)

private val SatisfiedResult: Satisfied = Satisfied(FoundSpaceLabel(safetyLabelType))

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val labels = featureMap(SpaceSafetyLabels).asInstanceOf[Seq[SpaceSafetyLabel]]

labels

.collectFirst {

case label if label.safetyLabelType == safetyLabelType =>

SatisfiedResult

}.getOrElse(UnsatisfiedResult)

}

}

case class MediaHasLabel(

safetyLabelType: MediaSafetyLabelType)

extends Condition

with HasSafetyLabelType {

override lazy val name: String = s"MediaHasLabel(${safetyLabelType.name})"

override val features: Set[Feature[\_]] = Set(MediaSafetyLabels)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

override val labelTypes: Set[SafetyLabelType] = Set(safetyLabelType)

private val UnsatisfiedResult: Unsatisfied = Unsatisfied(this)

private val SatisfiedResult: Satisfied = Satisfied(FoundMediaLabel(safetyLabelType))

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val labels = featureMap(MediaSafetyLabels).asInstanceOf[Seq[MediaSafetyLabel]]

labels

.collectFirst {

case label if label.safetyLabelType == safetyLabelType =>

SatisfiedResult

}.getOrElse(UnsatisfiedResult)

}

}

case class TweetHasLabelWithLanguageScoreAboveThreshold(

safetyLabel: TweetSafetyLabelType,

languagesToScoreThresholds: Map[String, Double])

extends Condition

with HasSafetyLabelType {

override lazy val name: String =

s"TweetHasLabelWithLanguageScoreAboveThreshold(${safetyLabel.name}, ${languagesToScoreThresholds.toString})"

override val features: Set[Feature[\_]] = Set(TweetSafetyLabels)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

override val labelTypes: Set[SafetyLabelType] = Set(safetyLabel)

private val UnsatisfiedResult: Unsatisfied = Unsatisfied(this)

private val SatisfiedResult: Satisfied =

Satisfied(

FoundTweetLabelWithLanguageScoreAboveThreshold(safetyLabel, languagesToScoreThresholds))

private[this] def isAboveThreshold(label: TweetSafetyLabel) = {

val isAboveThresholdOpt = for {

modelMetadata <- label.modelMetadata

calibratedLanguage <- modelMetadata.calibratedLanguage

threshold <- languagesToScoreThresholds.get(calibratedLanguage)

score <- label.score

} yield score >= threshold

isAboveThresholdOpt.getOrElse(false)

}

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val labels = featureMap(TweetSafetyLabels).asInstanceOf[Seq[TweetSafetyLabel]]

labels

.collectFirst {

case label

if label.labelType == safetyLabel

&& isAboveThreshold(label) =>

SatisfiedResult

}.getOrElse(UnsatisfiedResult)

}

}

case class TweetHasLabelWithScoreAboveThreshold(

safetyLabel: TweetSafetyLabelType,

threshold: Double)

extends Condition

with HasSafetyLabelType {

override lazy val name: String =

s"TweetHasLabelWithScoreAboveThreshold(${safetyLabel.name}, $threshold)"

override val features: Set[Feature[\_]] = Set(TweetSafetyLabels)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

override val labelTypes: Set[SafetyLabelType] = Set(safetyLabel)

private val UnsatisfiedResult = Unsatisfied(this)

private val SatisfiedResult =

Satisfied(FoundTweetLabelWithScoreAboveThreshold(safetyLabel, threshold))

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val labels = featureMap(TweetSafetyLabels).asInstanceOf[Seq[TweetSafetyLabel]]

labels

.collectFirst {

case label

if label.labelType == safetyLabel

&& label.score.exists(\_ >= threshold) =>

SatisfiedResult

}.getOrElse(UnsatisfiedResult)

}

}

case class TweetHasLabelWithScoreAboveThresholdWithParam(

safetyLabel: TweetSafetyLabelType,

thresholdParam: Param[Double])

extends Condition

with HasSafetyLabelType

with HasParams {

override lazy val name: String =

s"TweetHasLabelWithScoreAboveThreshold(${safetyLabel.name}, ${NamingUtils.getFriendlyName(thresholdParam)})"

override val features: Set[Feature[\_]] = Set(TweetSafetyLabels)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

override val labelTypes: Set[SafetyLabelType] = Set(safetyLabel)

private val UnsatisfiedResult = Unsatisfied(this)

override val params: Set[Param[\_]] = Set(thresholdParam)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val labels = featureMap(TweetSafetyLabels).asInstanceOf[Seq[TweetSafetyLabel]]

val threshold = evaluationContext.params(thresholdParam)

val SatisfiedResult =

Satisfied(FoundTweetLabelWithScoreAboveThreshold(safetyLabel, threshold))

labels

.collectFirst {

case label

if label.labelType == safetyLabel

&& label.score.exists(\_ >= threshold) =>

SatisfiedResult

}.getOrElse(UnsatisfiedResult)

}

}

case class TweetHasLabelWithLanguageIn(

safetyLabelType: TweetSafetyLabelType,

languages: Set[String])

extends Condition

with HasSafetyLabelType {

override lazy val name: String =

s"TweetHasLabelWithLanguageIn($safetyLabelType, $languages)"

override val features: Set[Feature[\_]] = Set(TweetSafetyLabels)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

override val labelTypes: Set[SafetyLabelType] = Set(safetyLabelType)

private val UnsatisfiedResult: Unsatisfied = Unsatisfied(this)

private val SatisfiedResult: Satisfied =

Satisfied(FoundTweetLabelWithLanguageIn(safetyLabelType, languages))

private[this] def hasLanguageMatch(label: TweetSafetyLabel): Boolean = {

val isMatchingLanguageOpt = for {

metadata <- label.modelMetadata

language <- metadata.calibratedLanguage

} yield languages.contains(language)

isMatchingLanguageOpt.getOrElse(false)

}

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

featureMap(TweetSafetyLabels)

.asInstanceOf[Seq[TweetSafetyLabel]]

.collectFirst {

case label if label.labelType == safetyLabelType && hasLanguageMatch(label) =>

SatisfiedResult

}.getOrElse(UnsatisfiedResult)

}

}

case class TweetHasLabelWithLanguagesWithParam(

safetyLabelType: TweetSafetyLabelType,

languageParam: Param[Seq[String]])

extends Condition

with HasSafetyLabelType

with HasParams {

override lazy val name: String =

s"TweetHasLabelWithLanguageIn($safetyLabelType, ${NamingUtils.getFriendlyName(languageParam)})"

override val features: Set[Feature[\_]] = Set(TweetSafetyLabels)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

override val labelTypes: Set[SafetyLabelType] = Set(safetyLabelType)

override val params: Set[Param[\_]] = Set(languageParam)

private val UnsatisfiedResult: Unsatisfied = Unsatisfied(this)

private[this] def hasLanguageMatch(label: TweetSafetyLabel, languages: Set[String]): Boolean = {

val isMatchingLanguageOpt = for {

metadata <- label.modelMetadata

language <- metadata.calibratedLanguage

} yield languages.contains(language)

isMatchingLanguageOpt.getOrElse(false)

}

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val languages = evaluationContext.params(languageParam).toSet

val SatisfiedResult: Satisfied =

Satisfied(FoundTweetLabelWithLanguageIn(safetyLabelType, languages))

featureMap(TweetSafetyLabels)

.asInstanceOf[Seq[TweetSafetyLabel]]

.collectFirst {

case label if label.labelType == safetyLabelType && hasLanguageMatch(label, languages) =>

SatisfiedResult

}.getOrElse(UnsatisfiedResult)

}

}

type TweetSafetyLabelPredicateFn = (TweetSafetyLabel) => Boolean

abstract class NamedTweetSafetyLabelPredicate(

private[rules] val fn: TweetSafetyLabelPredicateFn,

private[rules] val name: String)

abstract class TweetHasSafetyLabelWithPredicate(

private[rules] val safetyLabelType: TweetSafetyLabelType,

private[rules] val predicate: NamedTweetSafetyLabelPredicate)

extends Condition

with HasSafetyLabelType {

override lazy val name: String =

s"TweetHasSafetyLabelWithPredicate(${predicate.name}($safetyLabelType))"

override val features: Set[Feature[\_]] = Set(TweetSafetyLabels)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

override val labelTypes: Set[SafetyLabelType] = Set(safetyLabelType)

private val UnsatisfiedResult: Unsatisfied = Unsatisfied(this)

private val SatisfiedResult: Satisfied =

Satisfied(Result.FoundTweetSafetyLabelWithPredicate(safetyLabelType, predicate.name))

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

featureMap(TweetSafetyLabels)

.asInstanceOf[Seq[TweetSafetyLabel]]

.collectFirst {

case label if label.labelType == safetyLabelType && predicate.fn(label) =>

SatisfiedResult

}.getOrElse(UnsatisfiedResult)

}

}

object TweetHasSafetyLabelWithPredicate {

def unapply(

condition: TweetHasSafetyLabelWithPredicate

): Option[(TweetSafetyLabelType, NamedTweetSafetyLabelPredicate)] =

Some((condition.safetyLabelType, condition.predicate))

}

case class WithScoreEqInt(score: Int)

extends NamedTweetSafetyLabelPredicate(

fn = tweetSafetyLabel => tweetSafetyLabel.score.exists(s => s.intValue() == score),

name = "WithScoreEqInt"

)

case class TweetHasSafetyLabelWithScoreEqInt(

override val safetyLabelType: TweetSafetyLabelType,

score: Int)

extends TweetHasSafetyLabelWithPredicate(

safetyLabelType,

predicate = WithScoreEqInt(score)

)

case class TweetReplyToParentTweetBeforeDuration(duration: Duration) extends Condition {

override val features: Set[Feature[\_]] = Set(TweetParentId, TweetTimestamp)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult: Unsatisfied = Unsatisfied(this)

private val SatisfiedResult: Satisfied = Satisfied(

Result.IsTweetReplyToParentTweetBeforeDuration(duration))

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

featureMap

.get(TweetParentId).collect {

case tweetParentId: Long =>

featureMap

.get(TweetTimestamp).collect {

case tweetTimestamp: Time

if tweetTimestamp.diff(SnowflakeId.timeFromId(tweetParentId)) < duration =>

SatisfiedResult

}.getOrElse(UnsatisfiedResult)

}.getOrElse(UnsatisfiedResult)

}

}

case class TweetReplyToRootTweetBeforeDuration(duration: Duration) extends Condition {

override val features: Set[Feature[\_]] = Set(TweetConversationId, TweetTimestamp)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult: Unsatisfied = Unsatisfied(this)

private val SatisfiedResult: Satisfied = Satisfied(

Result.IsTweetReplyToRootTweetBeforeDuration(duration))

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

featureMap

.get(TweetConversationId).collect {

case tweetConversationId: Long =>

featureMap

.get(TweetTimestamp).collect {

case tweetTimestamp: Time

if tweetTimestamp.diff(

SnowflakeId.timeFromId(tweetConversationId)) < duration =>

SatisfiedResult

}.getOrElse(UnsatisfiedResult)

}.getOrElse(UnsatisfiedResult)

}

}

case class TweetComposedBefore(cutoffTimestamp: Time) extends Condition {

assert(cutoffTimestamp.inMilliseconds > SnowflakeId.FirstSnowflakeIdUnixTime)

override val features: Set[Feature[\_]] = Set(TweetTimestamp)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult: Unsatisfied = Unsatisfied(this)

private val SatisfiedResult: Satisfied = Satisfied(HasTweetTimestampBeforeCutoff)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

featureMap(TweetTimestamp) match {

case timestamp: Time if timestamp > cutoffTimestamp => UnsatisfiedResult

case \_ => SatisfiedResult

}

}

}

case class TweetComposedAfter(cutoffTimestamp: Time) extends Condition {

assert(cutoffTimestamp.inMilliseconds > SnowflakeId.FirstSnowflakeIdUnixTime)

override val features: Set[Feature[\_]] = Set(TweetTimestamp)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult: Unsatisfied = Unsatisfied(this)

private val SatisfiedResult: Satisfied = Satisfied(HasTweetTimestampAfterCutoff)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

featureMap(TweetTimestamp) match {

case timestamp: Time if timestamp > cutoffTimestamp => SatisfiedResult

case \_ => UnsatisfiedResult

}

}

}

case class TweetComposedAfterOffset(offset: Duration) extends Condition {

override val features: Set[Feature[\_]] = Set(TweetTimestamp)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult: Unsatisfied = Unsatisfied(this)

private val SatisfiedResult: Satisfied = Satisfied(HasTweetTimestampAfterOffset)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

featureMap(TweetTimestamp) match {

case timestamp: Time if timestamp > Time.now.minus(offset) => SatisfiedResult

case \_ => UnsatisfiedResult

}

}

}

case class TweetComposedAfterWithParam(cutoffTimeParam: Param[Time])

extends Condition

with HasParams {

override val features: Set[Feature[\_]] = Set(TweetTimestamp)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

override val params: Set[Param[\_]] = Set(cutoffTimeParam)

private val UnsatisfiedResult: Unsatisfied = Unsatisfied(this)

private val SatisfiedResult: Satisfied = Satisfied(HasTweetTimestampAfterCutoff)

override def preFilter(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): PreFilterResult = {

val cutoffTimestamp = evaluationContext.params(cutoffTimeParam)

if (cutoffTimestamp.inMilliseconds < SnowflakeId.FirstSnowflakeIdUnixTime) {

Filtered

} else {

super.preFilter(evaluationContext, featureMap)

}

}

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val cutoffTimestamp = evaluationContext.params(cutoffTimeParam)

featureMap(TweetTimestamp) match {

case \_: Time if cutoffTimestamp.inMilliseconds < SnowflakeId.FirstSnowflakeIdUnixTime =>

UnsatisfiedResult

case timestamp: Time if timestamp > cutoffTimestamp => SatisfiedResult

case \_ => UnsatisfiedResult

}

}

}

case class AuthorHasLabel(labelValue: UserLabelValue, shortCircuitable: Boolean = true)

extends Condition

with HasSafetyLabelType {

override lazy val name: String = s"AuthorHasLabel(${labelValue.name})"

override val features: Set[Feature[\_]] = Set(AuthorUserLabels)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

override val labelTypes: Set[SafetyLabelType] = Set(labelValue)

private val UnsatisfiedResult: Unsatisfied = Unsatisfied(this)

private val SatisfiedResult: Satisfied = Satisfied(FoundUserLabel(labelValue))

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val labels = featureMap(AuthorUserLabels).asInstanceOf[Seq[Label]].map(UserLabel.fromThrift)

labels

.collectFirst {

case label

if label.labelValue == labelValue

&& ExperimentBase.shouldFilterForSource(evaluationContext.params, label.source) =>

SatisfiedResult

}.getOrElse(UnsatisfiedResult)

}

}

abstract class ViewerHasRole(role: String) extends Condition {

override lazy val name: String = s"ViewerHasRole(${role})"

override val features: Set[Feature[\_]] = Set(ViewerRoles)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult: Unsatisfied = Unsatisfied(this)

private val SatisfiedResult: Satisfied = Satisfied(FoundUserRole(role))

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val roles = featureMap(ViewerRoles).asInstanceOf[Seq[String]]

if (roles.contains(role)) {

SatisfiedResult

} else {

UnsatisfiedResult

}

}

}

case object ViewerIsEmployee extends ViewerHasRole(ViewerRoles.EmployeeRole)

case class ViewerHasLabel(labelValue: UserLabelValue) extends Condition with HasSafetyLabelType {

override lazy val name: String = s"ViewerHasLabel(${labelValue.name})"

override val features: Set[Feature[\_]] = Set(ViewerUserLabels)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

override val labelTypes: Set[SafetyLabelType] = Set(labelValue)

private val UnsatisfiedResult: Unsatisfied = Unsatisfied(this)

private val SatisfiedResult: Satisfied = Satisfied(FoundUserLabel(labelValue))

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val labels = featureMap(ViewerUserLabels).asInstanceOf[Seq[Label]].map(UserLabel.fromThrift)

labels

.collectFirst {

case label

if label.labelValue == labelValue

&& ExperimentBase.shouldFilterForSource(evaluationContext.params, label.source) =>

SatisfiedResult

}.getOrElse(UnsatisfiedResult)

}

}

case object DeactivatedAuthor extends BooleanFeatureCondition(AuthorIsDeactivated)

case object ErasedAuthor extends BooleanFeatureCondition(AuthorIsErased)

case object OffboardedAuthor extends BooleanFeatureCondition(AuthorIsOffboarded)

case object ProtectedAuthor extends BooleanFeatureCondition(AuthorIsProtected)

case object VerifiedAuthor extends BooleanFeatureCondition(AuthorIsVerified)

case object NsfwUserAuthor extends BooleanFeatureCondition(AuthorIsNsfwUser)

case object NsfwAdminAuthor extends BooleanFeatureCondition(AuthorIsNsfwAdmin)

case object TweetHasNsfwUserAuthor extends BooleanFeatureCondition(TweetHasNsfwUser)

case object TweetHasNsfwAdminAuthor extends BooleanFeatureCondition(TweetHasNsfwAdmin)

case object TweetHasMedia extends BooleanFeatureCondition(TweetHasMediaFeature)

case object TweetHasDmcaMedia extends BooleanFeatureCondition(HasDmcaMediaFeature)

case object TweetHasCard extends BooleanFeatureCondition(TweetHasCardFeature)

case object IsPollCard extends BooleanFeatureCondition(CardIsPoll)

case object ProtectedViewer extends BooleanFeatureCondition(ViewerIsProtected)

case object SoftViewer extends BooleanFeatureCondition(ViewerIsSoftUser)

case object ViewerHasUqfEnabled

extends BooleanFeatureCondition(ViewerHasUniversalQualityFilterEnabled)

abstract class ViewerHasMatchingKeywordFor(muteSurface: MuteSurface) extends Condition {

override def features: Set[Feature[\_]] = Set(feature)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult = Unsatisfied(this)

private val feature: Feature[MutedKeyword] = muteSurface match {

case MuteSurface.HomeTimeline => ViewerMutesKeywordInTweetForHomeTimeline

case MuteSurface.Notifications => ViewerMutesKeywordInTweetForNotifications

case MuteSurface.TweetReplies => ViewerMutesKeywordInTweetForTweetReplies

case \_ => throw new NoSuchElementException(muteSurface.toString)

}

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val mutedKeyword = featureMap(feature)

.asInstanceOf[MutedKeyword]

Result.fromMutedKeyword(mutedKeyword, UnsatisfiedResult)

}

}

case object ViewerHasMatchingKeywordForHomeTimeline

extends ViewerHasMatchingKeywordFor(MuteSurface.HomeTimeline)

case object ViewerHasMatchingKeywordForNotifications

extends ViewerHasMatchingKeywordFor(MuteSurface.Notifications)

case object ViewerHasMatchingKeywordForTweetReplies

extends ViewerHasMatchingKeywordFor(MuteSurface.TweetReplies)

case object ViewerHasMatchingKeywordForAllSurfaces extends Condition {

override def features: Set[Feature[\_]] = Set(ViewerMutesKeywordInTweetForAllSurfaces)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val mutedKeyword = featureMap(ViewerMutesKeywordInTweetForAllSurfaces)

.asInstanceOf[MutedKeyword]

Result.fromMutedKeyword(mutedKeyword, UnsatisfiedResult)

}

}

abstract class ViewerHasMatchingKeywordInSpaceTitleFor(muteSurface: MuteSurface)

extends Condition {

override def features: Set[Feature[\_]] = Set(feature)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult = Unsatisfied(this)

private val feature: Feature[MutedKeyword] = muteSurface match {

case MuteSurface.Notifications => ViewerMutesKeywordInSpaceTitleForNotifications

case \_ => throw new NoSuchElementException(muteSurface.toString)

}

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val mutedKeyword = featureMap(feature)

.asInstanceOf[MutedKeyword]

Result.fromMutedKeyword(mutedKeyword, UnsatisfiedResult)

}

}

case object ViewerHasMatchingKeywordInSpaceTitleForNotifications

extends ViewerHasMatchingKeywordInSpaceTitleFor(MuteSurface.Notifications)

case object ViewerFiltersNoConfirmedEmail

extends BooleanFeatureCondition(

com.twitter.visibility.features.ViewerFiltersNoConfirmedEmail

)

case object ViewerFiltersNoConfirmedPhone

extends BooleanFeatureCondition(

com.twitter.visibility.features.ViewerFiltersNoConfirmedPhone

)

case object ViewerFiltersDefaultProfileImage

extends BooleanFeatureCondition(

com.twitter.visibility.features.ViewerFiltersDefaultProfileImage

)

case object ViewerFiltersNewUsers

extends BooleanFeatureCondition(

com.twitter.visibility.features.ViewerFiltersNewUsers

)

case object ViewerFiltersNotFollowedBy

extends BooleanFeatureCondition(

com.twitter.visibility.features.ViewerFiltersNotFollowedBy

)

case object AuthorHasConfirmedEmail

extends BooleanFeatureCondition(

com.twitter.visibility.features.AuthorHasConfirmedEmail

)

case object AuthorHasVerifiedPhone

extends BooleanFeatureCondition(

com.twitter.visibility.features.AuthorHasVerifiedPhone

)

case object AuthorHasDefaultProfileImage

extends BooleanFeatureCondition(

com.twitter.visibility.features.AuthorHasDefaultProfileImage

)

case object AuthorIsNewAccount extends Condition {

override val features: Set[Feature[\_]] = Set(AuthorAccountAge)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val age = featureMap(AuthorAccountAge).asInstanceOf[Duration]

if (age < 72.hours) {

Result.SatisfiedResult

} else {

UnsatisfiedResult

}

}

}

abstract class ViewerInJurisdiction extends Condition {

override def features: Set[Feature[\_]] = Set.empty

override val optionalFeatures: Set[Feature[\_]] = Set(RequestCountryCode, ViewerCountryCode)

protected val unsatisfiedResult = Unsatisfied(this)

protected case class CountryFeatures(

requestCountryCode: Option[String],

viewerCountryCode: Option[String])

def getCountryFeatures(featureMap: Map[Feature[\_], \_]): CountryFeatures = {

val requestCountryCodeOpt = featureMap

.get(RequestCountryCode)

.map(\_.asInstanceOf[String])

val viewerCountryCodeOpt = featureMap

.get(ViewerCountryCode)

.map(\_.asInstanceOf[String])

CountryFeatures(requestCountryCodeOpt, viewerCountryCodeOpt)

}

}

case class ViewerInHrcj(jurisdictions: Set[String]) extends ViewerInJurisdiction {

override def preFilter(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): PreFilterResult =

featureMap

.get(RequestCountryCode)

.map(\_.asInstanceOf[String])

.collectFirst {

case rcc if jurisdictions.contains(rcc) => NeedsFullEvaluation

}

.getOrElse(Filtered)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val countryFeatures = getCountryFeatures(featureMap)

countryFeatures match {

case CountryFeatures(Some(rcc), Some(vcc))

if jurisdictions.contains(rcc) && vcc.equals(rcc) =>

Satisfied(Result.ViewerInHrcj(rcc))

case \_ => unsatisfiedResult

}

}

}

case class ViewerOrRequestInJurisdiction(enabledCountriesParam: Param[Seq[String]])

extends ViewerInJurisdiction

with HasParams

with PreFilterOnOptionalFeatures {

override val params: Set[Param[\_]] = Set(enabledCountriesParam)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val countries: Seq[String] =

evaluationContext.params(enabledCountriesParam).map(c => c.toLowerCase)

val countryFeatures = getCountryFeatures(featureMap)

val countryCodeOpt =

countryFeatures.viewerCountryCode.orElse(countryFeatures.requestCountryCode)

countryCodeOpt match {

case Some(countryCode) if countries.contains(countryCode) =>

Satisfied(Result.ViewerOrRequestInCountry(countryCode))

case \_ => unsatisfiedResult

}

}

}

case class ViewerAgeInYearsGte(ageToCompare: Int, ignoreEmptyAge: Boolean = false)

extends Condition

with PreFilterOnOptionalFeatures {

override def features: Set[Feature[\_]] = Set.empty

override def optionalFeatures: Set[Feature[\_]] = Set(ViewerAge)

private val unsatisfiedResult = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result =

featureMap

.get(ViewerAge)

.map(\_.asInstanceOf[UserAge])

.collectFirst {

case UserAge(Some(age)) if age >= ageToCompare =>

Satisfied(Result.ViewerAgeInYears(age))

case UserAge(None) if ignoreEmptyAge =>

Satisfied(Result.NoViewerAge)

}

.getOrElse(unsatisfiedResult)

}

case class UnderageViewer(ageToCompare: Int) extends Condition with PreFilterOnOptionalFeatures {

override def features: Set[Feature[\_]] = Set.empty

override def optionalFeatures: Set[Feature[\_]] = Set(ViewerAge)

private val unsatisfiedResult = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result =

featureMap

.get(ViewerAge)

.map(\_.asInstanceOf[UserAge])

.collectFirst {

case UserAge(Some(age)) if age < ageToCompare => Satisfied(Result.ViewerAgeInYears(age))

}

.getOrElse(unsatisfiedResult)

}

case object ViewerMissingAge extends Condition with PreFilterOnOptionalFeatures {

override def features: Set[Feature[\_]] = Set.empty

override def optionalFeatures: Set[Feature[\_]] = Set(ViewerAge)

private val unsatisfiedResult = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result =

featureMap

.get(ViewerAge)

.map(\_.asInstanceOf[UserAge])

.collectFirst {

case UserAge(None) => Satisfied(Result.NoViewerAge)

}

.getOrElse(unsatisfiedResult)

}

case object ViewerOptInBlockingOnSearch extends BooleanFeatureCondition(ViewerOptInBlocking)

case object ViewerOptInFilteringOnSearch extends BooleanFeatureCondition(ViewerOptInFiltering)

case object SelfReply extends BooleanFeatureCondition(TweetIsSelfReply)

case object Nullcast extends BooleanFeatureCondition(TweetIsNullcast)

case object Moderated extends BooleanFeatureCondition(TweetIsModerated)

case object Retweet extends BooleanFeatureCondition(TweetIsRetweet)

case object IsFirstPageSearchResult extends Condition {

override val features: Set[Feature[\_]] = Set(SearchResultsPageNumber)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val searchResultsPageNumber = featureMap(SearchResultsPageNumber).asInstanceOf[Int]

if (searchResultsPageNumber == 1) {

Result.SatisfiedResult

} else {

UnsatisfiedResult

}

}

}

case object HasSearchCandidateCountGreaterThan45 extends Condition {

override val features: Set[Feature[\_]] = Set(SearchCandidateCount)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val searchCandidateCount = featureMap(SearchCandidateCount).asInstanceOf[Int]

if (searchCandidateCount > 45) {

Result.SatisfiedResult

} else {

UnsatisfiedResult

}

}

}

abstract class HasSearchQuerySource(querySourceToMatch: ThriftQuerySource) extends Condition {

override lazy val name: String = s"HasSearchQuerySource(${querySourceToMatch})"

override val features: Set[Feature[\_]] = Set(SearchQuerySource)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult = Unsatisfied(this)

private val SatisfiedResult: Satisfied = Satisfied(HasQuerySource(querySourceToMatch))

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val querySource = featureMap(SearchQuerySource).asInstanceOf[ThriftQuerySource]

if (querySourceToMatch.equals(querySource)) {

SatisfiedResult

} else {

UnsatisfiedResult

}

}

}

case object IsTrendClickSourceSearchResult extends Condition {

override val features: Set[Feature[\_]] = Set(SearchQuerySource)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult = Unsatisfied(this)

private def checkQuerySource[T](

featureMap: Map[Feature[\_], \_],

nonTrendSourceResult: T,

trendSourceResult: T

): T = {

val searchResultsPageNumber = featureMap(SearchQuerySource).asInstanceOf[ThriftQuerySource]

if (searchResultsPageNumber == ThriftQuerySource.TrendClick) {

trendSourceResult

} else {

nonTrendSourceResult

}

}

override def preFilter(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): PreFilterResult =

checkQuerySource(featureMap, Filtered, NotFiltered)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result =

checkQuerySource(featureMap, UnsatisfiedResult, Result.SatisfiedResult)

}

case object IsSearchHashtagClick extends HasSearchQuerySource(ThriftQuerySource.HashtagClick)

case object IsSearchTrendClick extends HasSearchQuerySource(ThriftQuerySource.TrendClick)

case object SearchQueryHasUser

extends BooleanFeatureCondition(com.twitter.visibility.features.SearchQueryHasUser)

case class Equals[T](feature: Feature[T], value: T) extends Condition {

override def features: Set[Feature[\_]] = Set(feature)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val SatisfiedResult: Result = Satisfied()

private val UnsatisfiedResult: Result = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val featureValue = featureMap(feature).asInstanceOf[T]

if (featureValue.equals(value)) SatisfiedResult else UnsatisfiedResult

}

}

case class FeatureEquals[T](left: Feature[T], right: Feature[T]) extends Condition {

override def features: Set[Feature[\_]] = Set.empty

override val optionalFeatures: Set[Feature[\_]] = Set(left, right)

private val SatisfiedResult: Result = Satisfied()

private val UnsatisfiedResult: Result = Unsatisfied(this)

override def preFilter(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): PreFilterResult = {

if (featureMap.contains(left) && featureMap.contains(right)) {

if (apply(evaluationContext, featureMap).asBoolean) {

NotFiltered

} else {

Filtered

}

} else {

NeedsFullEvaluation

}

}

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

if (featureMap.contains(left) && featureMap.contains(right)) {

val leftValue = featureMap(left).asInstanceOf[T]

val rightValue = featureMap(right).asInstanceOf[T]

if (leftValue.equals(rightValue)) SatisfiedResult else UnsatisfiedResult

} else {

UnsatisfiedResult

}

}

}

case class And(override val conditions: Condition\*)

extends Condition

with HasNestedConditions

with HasParams {

override lazy val name: String = s"(${conditions.map(\_.name).mkString(" And ")})"

override val features: Set[Feature[\_]] = conditions.flatMap(\_.features).toSet

override val optionalFeatures: Set[Feature[\_]] = conditions.flatMap(\_.optionalFeatures).toSet

override val params: Set[Param[\_]] =

conditions.collect { case p: HasParams => p.params }.flatten.toSet

override def preFilter(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): PreFilterResult = {

conditions.foldLeft(NotFiltered: PreFilterResult) {

case (NotFiltered, condition) => condition.preFilter(evaluationContext, featureMap)

case (Filtered, \_) => Filtered

case (NeedsFullEvaluation, condition) => {

condition.preFilter(evaluationContext, featureMap) match {

case Filtered => Filtered

case \_ => NeedsFullEvaluation

}

}

}

}

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

conditions.foldLeft(Result.SatisfiedResult) {

case (result @ Unsatisfied(\_), \_) => result

case (Result.SatisfiedResult, condition) => condition.apply(evaluationContext, featureMap)

case (result @ Satisfied(\_), condition) => {

condition.apply(evaluationContext, featureMap) match {

case r @ Unsatisfied(\_) => r

case \_ => result

}

}

}

}

}

case class Or(override val conditions: Condition\*)

extends Condition

with HasNestedConditions

with HasParams {

override lazy val name: String = s"(${conditions.map(\_.name).mkString(" Or ")})"

override val features: Set[Feature[\_]] = conditions.flatMap(\_.features).toSet

override val optionalFeatures: Set[Feature[\_]] = conditions.flatMap(\_.optionalFeatures).toSet

override val params: Set[Param[\_]] =

conditions.collect { case p: HasParams => p.params }.flatten.toSet

private val UnsatisfiedResult = Unsatisfied(this)

override def preFilter(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): PreFilterResult = {

conditions.foldLeft(Filtered: PreFilterResult) {

case (Filtered, c) => c.preFilter(evaluationContext, featureMap)

case (NotFiltered, \_) => NotFiltered

case (NeedsFullEvaluation, c) => {

c.preFilter(evaluationContext, featureMap) match {

case NotFiltered => NotFiltered

case \_ => NeedsFullEvaluation

}

}

}

}

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val foundSatisfiedCondition =

conditions.find(\_.apply(evaluationContext, featureMap).asBoolean)

if (foundSatisfiedCondition.isDefined) {

Result.SatisfiedResult

} else {

UnsatisfiedResult

}

}

}

case class Not(condition: Condition) extends Condition with HasNestedConditions with HasParams {

override lazy val name: String = s"Not(${condition.name})"

override val features: Set[Feature[\_]] = condition.features

override val optionalFeatures: Set[Feature[\_]] = condition.optionalFeatures

override val conditions: Seq[Condition] = Seq(condition)

override val params: Set[Param[\_]] =

conditions.collect { case p: HasParams => p.params }.flatten.toSet

private val UnsatisfiedResult = Unsatisfied(this)

override def preFilter(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): PreFilterResult =

condition.preFilter(evaluationContext, featureMap) match {

case Filtered => NotFiltered

case NotFiltered => Filtered

case \_ => NeedsFullEvaluation

}

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result =

if (condition(evaluationContext, featureMap).asBoolean) {

UnsatisfiedResult

} else {

Result.SatisfiedResult

}

}

val LoggedOutOrViewerNotFollowingAuthor: And =

And(NonAuthorViewer, Or(LoggedOutViewer, Not(ViewerDoesFollowAuthor)))

val LoggedOutOrViewerOptInFiltering: Or =

Or(LoggedOutViewer, ViewerOptInFilteringOnSearch)

val LoggedInViewer: Not = Not(LoggedOutViewer)

val OuterAuthorNotFollowingAuthor: And =

And(Not(OuterAuthorIsInnerAuthor), Not(OuterAuthorFollowsAuthor))

val IsFocalTweet: FeatureEquals[Long] = FeatureEquals(TweetId, FocalTweetId)

val NonHydratingConditions: Set[Class[\_]] = Set(

LoggedOutViewer,

NonAuthorViewer,

True,

TweetComposedAfter(Time.now),

TweetComposedBefore(Time.now)

).map(\_.getClass)

trait HasParams {

val params: Set[Param[\_]]

}

def hasLabelCondition(condition: Condition, tweetSafetyLabelType: TweetSafetyLabelType): Boolean =

condition match {

case lt: HasSafetyLabelType =>

lt.hasLabelType(tweetSafetyLabelType)

case \_ => false

}

def hasLabelCondition(condition: Condition, userLabelValue: UserLabelValue): Boolean =

condition match {

case lt: HasSafetyLabelType =>

lt.hasLabelType(userLabelValue)

case \_ => false

}

def hasLabelCondition(condition: Condition, spaceSafetyLabelType: SpaceSafetyLabelType): Boolean =

condition match {

case lt: HasSafetyLabelType =>

lt.hasLabelType(spaceSafetyLabelType)

case \_ => false

}

def hasLabelCondition(condition: Condition, mediaSafetyLabelType: MediaSafetyLabelType): Boolean =

condition match {

case lt: HasSafetyLabelType =>

lt.hasLabelType(mediaSafetyLabelType)

case \_ => false

}

case class Choose[T](

conditionMap: Map[T, Condition],

defaultCondition: Condition,

choiceParam: Param[T])

extends Condition

with HasNestedConditions

with HasParams {

override lazy val name: String =

s"(Either ${conditionMap.values.map(\_.name).mkString(", ")} or ${defaultCondition.name})"

override val features: Set[Feature[\_]] =

conditionMap.values.flatMap(\_.features).toSet ++ defaultCondition.features

override val optionalFeatures: Set[Feature[\_]] =

conditionMap.values.flatMap(\_.optionalFeatures).toSet ++ defaultCondition.optionalFeatures

override val conditions: Seq[Condition] = conditionMap.values.toSeq :+ defaultCondition

override val params: Set[Param[\_]] =

conditions.collect { case p: HasParams => p.params }.flatten.toSet

private[this] def getCondition(evaluationContext: EvaluationContext): Condition =

conditionMap.getOrElse(evaluationContext.params(choiceParam), defaultCondition)

override def preFilter(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): PreFilterResult =

getCondition(evaluationContext).preFilter(evaluationContext, featureMap)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result =

getCondition(evaluationContext)(evaluationContext, featureMap)

}

case class IfElse(

branchingCondition: Condition,

ifTrueCondition: Condition,

ifFalseCondition: Condition)

extends Condition

with HasNestedConditions

with HasParams {

override lazy val name: String =

s"(If ${branchingCondition.name} Then ${ifTrueCondition.name} Else ${ifFalseCondition.name})"

override val features: Set[Feature[\_]] =

branchingCondition.features ++ ifTrueCondition.features ++ ifFalseCondition.features

override val optionalFeatures: Set[Feature[\_]] =

branchingCondition.optionalFeatures ++ ifTrueCondition.optionalFeatures ++ ifFalseCondition.optionalFeatures

override val conditions: Seq[Condition] =

Seq(branchingCondition, ifTrueCondition, ifFalseCondition)

override val params: Set[Param[\_]] =

conditions.collect { case p: HasParams => p.params }.flatten.toSet

override def preFilter(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): PreFilterResult =

branchingCondition.preFilter(evaluationContext, featureMap) match {

case Filtered =>

ifFalseCondition.preFilter(evaluationContext, featureMap)

case NotFiltered =>

ifTrueCondition.preFilter(evaluationContext, featureMap)

case \_ =>

NeedsFullEvaluation

}

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result =

if (branchingCondition(evaluationContext, featureMap).asBoolean) {

ifTrueCondition(evaluationContext, featureMap)

} else {

ifFalseCondition(evaluationContext, featureMap)

}

}

case class GatedAlternate[T](

defaultCondition: Condition,

alternateConditions: Map[T, Condition],

bucketIdentifierToUseOnDisagreementParam: Param[Option[T]])

extends Condition

with HasNestedConditions

with HasParams {

override lazy val name: String =

s"(${defaultCondition.name} or sometimes ${alternateConditions.values.map(\_.name).mkString(" or ")})"

override val features: Set[Feature[\_]] =

defaultCondition.features ++ alternateConditions.values.flatMap(\_.features)

override val optionalFeatures: Set[Feature[\_]] =

defaultCondition.optionalFeatures ++ alternateConditions.values.flatMap(\_.optionalFeatures)

override val conditions: Seq[Condition] = Seq(defaultCondition) ++ alternateConditions.values

override val params: Set[Param[\_]] =

conditions.collect { case p: HasParams => p.params }.flatten.toSet

override def preFilter(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): PreFilterResult =

if (defaultCondition.preFilter(evaluationContext, featureMap) == Filtered &&

alternateConditions.values.forall(\_.preFilter(evaluationContext, featureMap) == Filtered)) {

Filtered

} else if (defaultCondition.preFilter(evaluationContext, featureMap) == NotFiltered &&

alternateConditions.values.forall(

\_.preFilter(evaluationContext, featureMap) == NotFiltered)) {

NotFiltered

} else {

NeedsFullEvaluation

}

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val defaultConditionResult: Result = defaultCondition(evaluationContext, featureMap)

val alternateConditionResult: Map[T, Result] =

alternateConditions.mapValues(\_(evaluationContext, featureMap))

if (alternateConditionResult.values.exists(\_.asBoolean != defaultConditionResult.asBoolean)) {

evaluationContext.params(bucketIdentifierToUseOnDisagreementParam) match {

case Some(bucket) if alternateConditionResult.contains(bucket) =>

alternateConditionResult(bucket)

case \_ =>

defaultConditionResult

}

} else {

defaultConditionResult

}

}

}

case class EnumGatedAlternate[E <: Enumeration](

defaultCondition: Condition,

alternateConditions: Map[E#Value, Condition],

bucketIdentifierToUseOnDisagreementParam: EnumParam[E])

extends Condition

with HasNestedConditions

with HasParams {

override lazy val name: String =

s"(${defaultCondition.name} or sometimes ${alternateConditions.values.map(\_.name).mkString(" or ")})"

override val features: Set[Feature[\_]] =

defaultCondition.features ++ alternateConditions.values.flatMap(\_.features)

override val optionalFeatures: Set[Feature[\_]] =

defaultCondition.optionalFeatures ++ alternateConditions.values.flatMap(\_.optionalFeatures)

override val conditions: Seq[Condition] = Seq(defaultCondition) ++ alternateConditions.values

override val params: Set[Param[\_]] =

conditions

.collect {

case p: HasParams => p.params

}.flatten.toSet + bucketIdentifierToUseOnDisagreementParam

override def preFilter(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): PreFilterResult =

if (defaultCondition.preFilter(evaluationContext, featureMap) == Filtered &&

alternateConditions.values.forall(\_.preFilter(evaluationContext, featureMap) == Filtered)) {

Filtered

} else if (defaultCondition.preFilter(evaluationContext, featureMap) == NotFiltered &&

alternateConditions.values.forall(

\_.preFilter(evaluationContext, featureMap) == NotFiltered)) {

NotFiltered

} else {

NeedsFullEvaluation

}

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val defaultConditionResult: Result = defaultCondition(evaluationContext, featureMap)

val alternateConditionResult: Map[E#Value, Result] =

alternateConditions.mapValues(\_(evaluationContext, featureMap))

if (alternateConditionResult.values.exists(\_.asBoolean != defaultConditionResult.asBoolean)) {

evaluationContext.params(bucketIdentifierToUseOnDisagreementParam) match {

case bucket if alternateConditionResult.contains(bucket) =>

alternateConditionResult(bucket)

case \_ =>

defaultConditionResult

}

} else {

defaultConditionResult

}

}

}

case object IsTestTweet extends Condition {

override val features: Set[Feature[\_]] = Set(TweetId)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

if (!featureMap.contains(TweetId)) {

UnsatisfiedResult

} else {

Result.SatisfiedResult

}

}

}

case object IsTweetInTweetLevelStcmHoldback extends Condition {

override val features: Set[Feature[\_]] = Set(TweetId)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val tweetId: Long = featureMap(TweetId).asInstanceOf[Long]

if (StcmTweetHoldback.isTweetInHoldback(tweetId)) {

Result.SatisfiedResult

} else {

UnsatisfiedResult

}

}

}

case object MediaRestrictedInViewerCountry extends Condition {

override val features: Set[Feature[\_]] =

Set(MediaGeoRestrictionsAllowList, MediaGeoRestrictionsDenyList)

override val optionalFeatures: Set[Feature[\_]] = Set(RequestCountryCode)

private val UnsatisfiedResult = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val requestCountryCode = TakedownReasons.normalizeCountryCodeOption(

featureMap.get(RequestCountryCode).asInstanceOf[Option[String]])

val allowlistCountryCodes =

featureMap(MediaGeoRestrictionsAllowList).asInstanceOf[Seq[String]]

val denylistCountryCodes =

featureMap(MediaGeoRestrictionsDenyList).asInstanceOf[Seq[String]]

if ((allowlistCountryCodes.nonEmpty && !allowlistCountryCodes.contains(requestCountryCode))

|| denylistCountryCodes.contains(requestCountryCode)) {

Result.SatisfiedResult

} else {

UnsatisfiedResult

}

}

}

case object OneToOneDmConversation

extends BooleanFeatureCondition(DmConversationIsOneToOneConversation)

case object DmConversationTimelineIsEmpty

extends BooleanFeatureCondition(DmConversationHasEmptyTimeline)

case object DmConversationLastReadableEventIdIsValid

extends BooleanFeatureCondition(DmConversationHasValidLastReadableEventId)

case object ViewerIsDmConversationParticipant

extends BooleanFeatureCondition(feats.ViewerIsDmConversationParticipant)

case object DmConversationInfoExists

extends BooleanFeatureCondition(feats.DmConversationInfoExists)

case object DmConversationTimelineExists

extends BooleanFeatureCondition(feats.DmConversationTimelineExists)

case object DmEventIsBeforeLastClearedEvent

extends BooleanFeatureCondition(DmEventOccurredBeforeLastClearedEvent)

case object DmEventIsBeforeJoinConversationEvent

extends BooleanFeatureCondition(DmEventOccurredBeforeJoinConversationEvent)

case object DmEventIsDeleted extends BooleanFeatureCondition(feats.DmEventIsDeleted)

case object DmEventIsHidden extends BooleanFeatureCondition(feats.DmEventIsHidden)

case object ViewerIsDmEventInitiatingUser

extends BooleanFeatureCondition(feats.ViewerIsDmEventInitiatingUser)

case object DmEventInOneToOneConversationWithUnavailableUser

extends BooleanFeatureCondition(feats.DmEventInOneToOneConversationWithUnavailableUser)

case object DmEventInOneToOneConversation

extends BooleanFeatureCondition(feats.DmEventInOneToOneConversation)

case object MessageCreateDmEvent extends BooleanFeatureCondition(DmEventIsMessageCreateEvent)

case object WelcomeMessageCreateDmEvent

extends BooleanFeatureCondition(DmEventIsWelcomeMessageCreateEvent)

case object LastMessageReadUpdateDmEvent

extends BooleanFeatureCondition(DmEventIsLastMessageReadUpdateEvent)

case object JoinConversationDmEvent

extends BooleanFeatureCondition(DmEventIsJoinConversationEvent)

case object ConversationCreateDmEvent

extends BooleanFeatureCondition(DmEventIsConversationCreateEvent)

case object TrustConversationDmEvent

extends BooleanFeatureCondition(DmEventIsTrustConversationEvent)

case object CsFeedbackSubmittedDmEvent

extends BooleanFeatureCondition(DmEventIsCsFeedbackSubmitted)

case object CsFeedbackDismissedDmEvent

extends BooleanFeatureCondition(DmEventIsCsFeedbackDismissed)

case object PerspectivalJoinConversationDmEvent

extends BooleanFeatureCondition(feats.DmEventIsPerspectivalJoinConversationEvent)

case class SpaceHasLabelWithScoreAboveThresholdWithParam(

spaceSafetyLabelType: SpaceSafetyLabelType,

thresholdParam: Param[Double])

extends Condition

with HasParams {

override lazy val name: String =

s"SpaceHasLabelWithScoreAboveThreshold(${spaceSafetyLabelType.name}, ${NamingUtils.getFriendlyName(thresholdParam)})"

override val features: Set[Feature[\_]] = Set(SpaceSafetyLabels)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult = Unsatisfied(this)

override val params: Set[Param[\_]] = Set(thresholdParam)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val labels = featureMap(SpaceSafetyLabels).asInstanceOf[Seq[SpaceSafetyLabel]]

val threshold = evaluationContext.params(thresholdParam)

val SatisfiedResult =

Satisfied(FoundSpaceLabelWithScoreAboveThreshold(spaceSafetyLabelType, threshold))

labels

.collectFirst {

case label

if label.safetyLabelType == spaceSafetyLabelType

&& label.safetyLabel.score.exists(\_ >= threshold) =>

SatisfiedResult

}.getOrElse(UnsatisfiedResult)

}

}

case class CardUriHasRootDomain(rootDomainParam: Param[Seq[String]])

extends Condition

with HasParams {

override lazy val name: String =

s"CardUriHasRootDomain(${NamingUtils.getFriendlyName(rootDomainParam)})"

override val features: Set[Feature[\_]] = Set(CardUriHost)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult = Unsatisfied(this)

override val params: Set[Param[\_]] = Set(rootDomainParam)

private[this] def isHostDomainOrSubdomain(domain: String, host: String): Boolean =

host == domain || host.endsWith("." + domain)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val cardUriHost = featureMap(CardUriHost).asInstanceOf[String]

val rootDomains = evaluationContext.params(rootDomainParam)

if (rootDomains.exists(isHostDomainOrSubdomain(\_, cardUriHost))) {

Satisfied(FoundCardUriRootDomain(cardUriHost))

} else {

UnsatisfiedResult

}

}

}

case class TweetHasViolationOfLevel(level: ViolationLevel)

extends Condition

with HasSafetyLabelType {

override lazy val name: String = s"tweetHasViolationOfLevel(${level})"

override val features: Set[Feature[\_]] = Set(TweetSafetyLabels)

override def optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult: Unsatisfied = Unsatisfied(this)

private val SatisfiedResult: Satisfied = Satisfied(FoundTweetViolationOfLevel(level))

override val labelTypes: Set[SafetyLabelType] =

ViolationLevel.violationLevelToSafetyLabels

.getOrElse(level, Set.empty)

.map(\_.asInstanceOf[SafetyLabelType])

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val labels = featureMap(TweetSafetyLabels).asInstanceOf[Seq[TweetSafetyLabel]]

if (labels.map(ViolationLevel.fromTweetSafetyLabel).contains(level)) {

SatisfiedResult

} else {

UnsatisfiedResult

}

}

}

case object TweetHasViolationOfAnyLevel extends Condition with HasSafetyLabelType {

override lazy val name: String = s"tweetHasViolationOfAnyLevel"

override val features: Set[Feature[\_]] = Set(TweetSafetyLabels)

override def optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult: Unsatisfied = Unsatisfied(this)

private val SatisfiedResult: Satisfied = Satisfied(FoundTweetViolationOfSomeLevel)

override val labelTypes: Set[SafetyLabelType] =

ViolationLevel.violationLevelToSafetyLabels.values

.reduceLeft(\_ ++ \_)

.map(\_.asInstanceOf[SafetyLabelType])

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val labels = featureMap(TweetSafetyLabels).asInstanceOf[Seq[TweetSafetyLabel]]

if (labels

.map(ViolationLevel.fromTweetSafetyLabelOpt).collect {

case Some(level) => level

}.nonEmpty) {

SatisfiedResult

} else {

UnsatisfiedResult

}

}

}

case object TweetIsEditTweet extends BooleanFeatureCondition(TweetIsEditTweetFeature)

case object TweetIsStaleTweet extends BooleanFeatureCondition(TweetIsStaleTweetFeature)

case class ViewerHasAdultMediaSettingLevel(settingLevelToCompare: SensitiveMediaSettingsLevel)

extends Condition {

override def features: Set[Feature[\_]] = Set(ViewerSensitiveMediaSettings)

override def optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

featureMap

.get(ViewerSensitiveMediaSettings)

.map(\_.asInstanceOf[UserSensitiveMediaSettings])

.collectFirst {

case UserSensitiveMediaSettings(Some(setting))

if (setting.viewAdultContent == settingLevelToCompare) =>

Result.SatisfiedResult

case UserSensitiveMediaSettings(None) => UnsatisfiedResult

}.getOrElse(UnsatisfiedResult)

}

}

case class ViewerHasViolentMediaSettingLevel(settingLevelToCompare: SensitiveMediaSettingsLevel)

extends Condition {

override def features: Set[Feature[\_]] = Set(ViewerSensitiveMediaSettings)

override def optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

featureMap

.get(ViewerSensitiveMediaSettings)

.map(\_.asInstanceOf[UserSensitiveMediaSettings])

.collectFirst {

case UserSensitiveMediaSettings(Some(setting))

if (setting.viewViolentContent == settingLevelToCompare) =>

Result.SatisfiedResult

case UserSensitiveMediaSettings(None) => UnsatisfiedResult

}.getOrElse(UnsatisfiedResult)

}

}

case class ViewerHasOtherSensitiveMediaSettingLevel(

settingLevelToCompare: SensitiveMediaSettingsLevel)

extends Condition {

override def features: Set[Feature[\_]] = Set(ViewerSensitiveMediaSettings)

override def optionalFeatures: Set[Feature[\_]] = Set.empty

private val UnsatisfiedResult = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

featureMap

.get(ViewerSensitiveMediaSettings)

.map(\_.asInstanceOf[UserSensitiveMediaSettings])

.collectFirst {

case UserSensitiveMediaSettings(Some(setting))

if (setting.viewOtherContent == settingLevelToCompare) =>

Result.SatisfiedResult

case UserSensitiveMediaSettings(None) => UnsatisfiedResult

}.getOrElse(UnsatisfiedResult)

}

}

private[rules] val ToxrfTweetFilteredForAuthor =

Equals(ToxicReplyFilterState, FilterState.FilteredFromAuthor)

private[rules] case object ToxrfViewerIsConversationAuthor

extends BooleanFeatureCondition(ToxicReplyFilterConversationAuthorIsViewer)

val ToxrfFilteredFromAuthorViewer =

And(LoggedInViewer, ToxrfTweetFilteredForAuthor, ToxrfViewerIsConversationAuthor)

case object SearchQueryMatchesScreenName extends Condition {

override def features: Set[Feature[\_]] = Set.empty

override def optionalFeatures: Set[Feature[\_]] = Set(RawQuery, AuthorScreenName)

private val UnsatisfiedResult = Unsatisfied(this)

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

if (featureMap.contains(RawQuery) && featureMap.contains(AuthorScreenName)) {

val rawQuery = featureMap(RawQuery).asInstanceOf[String]

val authorScreenName = featureMap(AuthorScreenName).asInstanceOf[String]

if (rawQuery.equalsIgnoreCase(authorScreenName)) {

Result.SatisfiedResult

} else {

UnsatisfiedResult

}

} else {

UnsatisfiedResult

}

}

}

object SearchQueryDoesNotMatchScreenNameConditionBuilder {

def apply(condition: Condition, ruleParam: RuleParam[Boolean]): Choose[Boolean] = {

Choose(

conditionMap =

Map(true -> And(condition, Not(SearchQueryMatchesScreenName)), false -> condition),

defaultCondition = condition,

choiceParam = ruleParam

)

}

}

val SearchQueryDoesNotMatchScreenNameDefaultTrueCondition: Choose[Boolean] =

SearchQueryDoesNotMatchScreenNameConditionBuilder(Condition.True, RuleParams.False)

case object OptionalNonAuthorViewer extends Condition {

override val features: Set[Feature[\_]] = Set()

override val optionalFeatures: Set[Feature[\_]] = Set(AuthorId, ViewerId)

private val UnsatisfiedResult = Unsatisfied(this)

override def preFilter(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): PreFilterResult = {

NeedsFullEvaluation

}

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val authorIdsOpt = featureMap.get(AuthorId).asInstanceOf[Option[Set[Long]]]

val viewerIdOpt = featureMap.get(ViewerId).asInstanceOf[Option[Long]]

(for {

authorIds <- authorIdsOpt

viewerId <- viewerIdOpt

} yield {

if (authorIds.contains(viewerId)) UnsatisfiedResult

else Result.SatisfiedResult

}) getOrElse {

Result.SatisfiedResult

}

}

}

case class ViewerLocatedInApplicableCountriesOfMediaWithholdingLabel(

safetyLabelType: MediaSafetyLabelType)

extends ViewerInJurisdiction

with HasSafetyLabelType {

override lazy val name: String =

s"ViewerLocatedInApplicableCountriesOfMediaLabel(${safetyLabelType.name})"

override val features: Set[Feature[\_]] = Set(MediaSafetyLabels)

override val optionalFeatures: Set[Feature[\_]] = Set(ViewerCountryCode, RequestCountryCode)

override val labelTypes: Set[SafetyLabelType] = Set(safetyLabelType)

private val UnsatisfiedResult: Unsatisfied = Unsatisfied(this)

private[this] def isInApplicableCountries(

countryCodeOpt: Option[String],

label: SafetyLabel

): Boolean = {

val inApplicableCountry = (for {

applicableCountries <- label.applicableCountries

countryCode <- countryCodeOpt

} yield {

applicableCountries.contains(countryCode)

}) getOrElse (false)

inApplicableCountry

}

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val labels = featureMap(MediaSafetyLabels).asInstanceOf[Seq[MediaSafetyLabel]]

val countryFeatures = getCountryFeatures(featureMap)

val countryCodeOpt = countryFeatures.requestCountryCode

.orElse(countryFeatures.viewerCountryCode)

labels

.collectFirst {

case label

if label.safetyLabelType == safetyLabelType

&&

isInApplicableCountries(countryCodeOpt, label.safetyLabel) =>

Result.SatisfiedResult

}.getOrElse(UnsatisfiedResult)

}

}

case class MediaHasLabelWithWorldwideWithholding(safetyLabelType: MediaSafetyLabelType)

extends Condition

with HasSafetyLabelType {

override lazy val name: String =

s"MediaHasLabelWithWorldwideWithholding(${safetyLabelType.name})"

override val features: Set[Feature[\_]] = Set(MediaSafetyLabels)

override val optionalFeatures: Set[Feature[\_]] = Set.empty

override val labelTypes: Set[SafetyLabelType] = Set(safetyLabelType)

private val UnsatisfiedResult: Unsatisfied = Unsatisfied(this)

private[this] def isWithheldWorldwide(label: SafetyLabel): Boolean = {

label.applicableCountries.map(\_.contains("xx")).getOrElse(false)

}

override def apply(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): Result = {

val labels = featureMap(MediaSafetyLabels).asInstanceOf[Seq[MediaSafetyLabel]]

labels

.collectFirst {

case label

if label.safetyLabelType == safetyLabelType

&& isWithheldWorldwide(label.safetyLabel) =>

Result.SatisfiedResult

}.getOrElse(UnsatisfiedResult)

}

}

}