package com.twitter.tsp.handlers

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.mux.ClientDiscardedRequestException

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.util.StatsUtil

import com.twitter.simclusters\_v2.common.SemanticCoreEntityId

import com.twitter.simclusters\_v2.common.TweetId

import com.twitter.simclusters\_v2.thriftscala.EmbeddingType

import com.twitter.simclusters\_v2.thriftscala.ModelVersion

import com.twitter.strato.response.Err

import com.twitter.storehaus.ReadableStore

import com.twitter.timelines.configapi.Params

import com.twitter.topic\_recos.common.Configs.ConsumerTopicEmbeddingType

import com.twitter.topic\_recos.common.Configs.DefaultModelVersion

import com.twitter.topic\_recos.common.Configs.ProducerTopicEmbeddingType

import com.twitter.topic\_recos.common.Configs.TweetEmbeddingType

import com.twitter.topiclisting.TopicListingViewerContext

import com.twitter.topic\_recos.common.LocaleUtil

import com.twitter.topiclisting.AnnotationRuleProvider

import com.twitter.tsp.common.DeciderConstants

import com.twitter.tsp.common.LoadShedder

import com.twitter.tsp.common.RecTargetFactory

import com.twitter.tsp.common.TopicSocialProofDecider

import com.twitter.tsp.common.TopicSocialProofParams

import com.twitter.tsp.stores.TopicSocialProofStore

import com.twitter.tsp.stores.TopicSocialProofStore.TopicSocialProof

import com.twitter.tsp.stores.UttTopicFilterStore

import com.twitter.tsp.stores.TopicTweetsCosineSimilarityAggregateStore.ScoreKey

import com.twitter.tsp.thriftscala.MetricTag

import com.twitter.tsp.thriftscala.TopicFollowType

import com.twitter.tsp.thriftscala.TopicListingSetting

import com.twitter.tsp.thriftscala.TopicSocialProofRequest

import com.twitter.tsp.thriftscala.TopicSocialProofResponse

import com.twitter.tsp.thriftscala.TopicWithScore

import com.twitter.tsp.thriftscala.TspTweetInfo

import com.twitter.tsp.utils.HealthSignalsUtils

import com.twitter.util.Future

import com.twitter.util.Timer

import com.twitter.util.Duration

import com.twitter.util.TimeoutException

import scala.util.Random

class TopicSocialProofHandler(

topicSocialProofStore: ReadableStore[TopicSocialProofStore.Query, Seq[TopicSocialProof]],

tweetInfoStore: ReadableStore[TweetId, TspTweetInfo],

uttTopicFilterStore: UttTopicFilterStore,

recTargetFactory: RecTargetFactory,

decider: TopicSocialProofDecider,

statsReceiver: StatsReceiver,

loadShedder: LoadShedder,

timer: Timer) {

import TopicSocialProofHandler.\_

def getTopicSocialProofResponse(

request: TopicSocialProofRequest

): Future[TopicSocialProofResponse] = {

val scopedStats = statsReceiver.scope(request.displayLocation.toString)

scopedStats.counter("fanoutRequests").incr(request.tweetIds.size)

scopedStats.stat("numTweetsPerRequest").add(request.tweetIds.size)

StatsUtil.trackBlockStats(scopedStats) {

recTargetFactory

.buildRecTopicSocialProofTarget(request).flatMap { target =>

val enableCosineSimilarityScoreCalculation =

decider.isAvailable(DeciderConstants.enableTopicSocialProofScore)

val semanticCoreVersionId =

target.params(TopicSocialProofParams.TopicTweetsSemanticCoreVersionId)

val semanticCoreVersionIdsSet =

target.params(TopicSocialProofParams.TopicTweetsSemanticCoreVersionIdsSet)

val allowListWithTopicFollowTypeFut = uttTopicFilterStore

.getAllowListTopicsForUser(

request.userId,

request.topicListingSetting,

TopicListingViewerContext

.fromThrift(request.context).copy(languageCode =

LocaleUtil.getStandardLanguageCode(request.context.languageCode)),

request.bypassModes.map(\_.toSet)

).rescue {

case \_ =>

scopedStats.counter("uttTopicFilterStoreFailure").incr()

Future.value(Map.empty[SemanticCoreEntityId, Option[TopicFollowType]])

}

val tweetInfoMapFut: Future[Map[TweetId, Option[TspTweetInfo]]] = Future

.collect(

tweetInfoStore.multiGet(request.tweetIds.toSet)

).raiseWithin(TweetInfoStoreTimeout)(timer).rescue {

case \_: TimeoutException =>

scopedStats.counter("tweetInfoStoreTimeout").incr()

Future.value(Map.empty[TweetId, Option[TspTweetInfo]])

case \_ =>

scopedStats.counter("tweetInfoStoreFailure").incr()

Future.value(Map.empty[TweetId, Option[TspTweetInfo]])

}

val definedTweetInfoMapFut =

keepTweetsWithTweetInfoAndLanguage(tweetInfoMapFut, request.displayLocation.toString)

Future

.join(definedTweetInfoMapFut, allowListWithTopicFollowTypeFut).map {

case (tweetInfoMap, allowListWithTopicFollowType) =>

val tweetIdsToQuery = tweetInfoMap.keys.toSet

val topicProofQueries =

tweetIdsToQuery.map { tweetId =>

TopicSocialProofStore.Query(

TopicSocialProofStore.CacheableQuery(

tweetId = tweetId,

tweetLanguage = LocaleUtil.getSupportedStandardLanguageCodeWithDefault(

tweetInfoMap.getOrElse(tweetId, None).flatMap {

\_.language

}),

enableCosineSimilarityScoreCalculation =

enableCosineSimilarityScoreCalculation

),

allowedSemanticCoreVersionIds = semanticCoreVersionIdsSet

)

}

val topicSocialProofsFut: Future[Map[TweetId, Seq[TopicSocialProof]]] = {

Future

.collect(topicSocialProofStore.multiGet(topicProofQueries)).map(\_.map {

case (query, results) =>

query.cacheableQuery.tweetId -> results.toSeq.flatten.filter(

\_.semanticCoreVersionId == semanticCoreVersionId)

})

}.raiseWithin(TopicSocialProofStoreTimeout)(timer).rescue {

case \_: TimeoutException =>

scopedStats.counter("topicSocialProofStoreTimeout").incr()

Future(Map.empty[TweetId, Seq[TopicSocialProof]])

case \_ =>

scopedStats.counter("topicSocialProofStoreFailure").incr()

Future(Map.empty[TweetId, Seq[TopicSocialProof]])

}

val random = new Random(seed = request.userId.toInt)

topicSocialProofsFut.map { topicSocialProofs =>

val filteredTopicSocialProofs = filterByAllowedList(

topicSocialProofs,

request.topicListingSetting,

allowListWithTopicFollowType.keySet

)

val filteredTopicSocialProofsEmptyCount: Int =

filteredTopicSocialProofs.count {

case (\_, topicSocialProofs: Seq[TopicSocialProof]) =>

topicSocialProofs.isEmpty

}

scopedStats

.counter("filteredTopicSocialProofsCount").incr(filteredTopicSocialProofs.size)

scopedStats

.counter("filteredTopicSocialProofsEmptyCount").incr(

filteredTopicSocialProofsEmptyCount)

if (isCrTopicTweets(request)) {

val socialProofs = filteredTopicSocialProofs.mapValues(\_.flatMap { topicProof =>

val topicWithScores = buildTopicWithRandomScore(

topicProof,

allowListWithTopicFollowType,

random

)

topicWithScores

})

TopicSocialProofResponse(socialProofs)

} else {

val socialProofs = filteredTopicSocialProofs.mapValues(\_.flatMap { topicProof =>

getTopicProofScore(

topicProof = topicProof,

allowListWithTopicFollowType = allowListWithTopicFollowType,

params = target.params,

random = random,

statsReceiver = statsReceiver

)

}.sortBy(-\_.score).take(MaxCandidates))

val personalizedContextSocialProofs =

if (target.params(TopicSocialProofParams.EnablePersonalizedContextTopics)) {

val personalizedContextEligibility =

checkPersonalizedContextsEligibility(

target.params,

allowListWithTopicFollowType)

val filteredTweets =

filterPersonalizedContexts(socialProofs, tweetInfoMap, target.params)

backfillPersonalizedContexts(

allowListWithTopicFollowType,

filteredTweets,

request.tags.getOrElse(Map.empty),

personalizedContextEligibility)

} else {

Map.empty[TweetId, Seq[TopicWithScore]]

}

val mergedSocialProofs = socialProofs.map {

case (tweetId, proofs) =>

(

tweetId,

proofs

++ personalizedContextSocialProofs.getOrElse(tweetId, Seq.empty))

}

// Note that we will NOT filter out tweets with no TSP in either case

TopicSocialProofResponse(mergedSocialProofs)

}

}

}

}.flatten.raiseWithin(Timeout)(timer).rescue {

case \_: ClientDiscardedRequestException =>

scopedStats.counter("ClientDiscardedRequestException").incr()

Future.value(DefaultResponse)

case err: Err if err.code == Err.Cancelled =>

scopedStats.counter("CancelledErr").incr()

Future.value(DefaultResponse)

case \_ =>

scopedStats.counter("FailedRequests").incr()

Future.value(DefaultResponse)

}

}

}

/\*\*

\* Fetch the Score for each Topic Social Proof

\*/

private def getTopicProofScore(

topicProof: TopicSocialProof,

allowListWithTopicFollowType: Map[SemanticCoreEntityId, Option[TopicFollowType]],

params: Params,

random: Random,

statsReceiver: StatsReceiver

): Option[TopicWithScore] = {

val scopedStats = statsReceiver.scope("getTopicProofScores")

val enableTweetToTopicScoreRanking =

params(TopicSocialProofParams.EnableTweetToTopicScoreRanking)

val minTweetToTopicCosineSimilarityThreshold =

params(TopicSocialProofParams.TweetToTopicCosineSimilarityThreshold)

val topicWithScore =

if (enableTweetToTopicScoreRanking) {

scopedStats.counter("enableTweetToTopicScoreRanking").incr()

buildTopicWithValidScore(

topicProof,

TweetEmbeddingType,

Some(ConsumerTopicEmbeddingType),

Some(ProducerTopicEmbeddingType),

allowListWithTopicFollowType,

DefaultModelVersion,

minTweetToTopicCosineSimilarityThreshold

)

} else {

scopedStats.counter("buildTopicWithRandomScore").incr()

buildTopicWithRandomScore(

topicProof,

allowListWithTopicFollowType,

random

)

}

topicWithScore

}

private[handlers] def isCrTopicTweets(

request: TopicSocialProofRequest

): Boolean = {

// CrTopic (across a variety of DisplayLocations) is the only use case with TopicListingSetting.All

request.topicListingSetting == TopicListingSetting.All

}

/\*\*

\* Consolidate logics relevant to whether only quality topics should be enabled for Implicit Follows

\*/

/\*\*\*

\* Consolidate logics relevant to whether Personalized Contexts backfilling should be enabled

\*/

private[handlers] def checkPersonalizedContextsEligibility(

params: Params,

allowListWithTopicFollowType: Map[SemanticCoreEntityId, Option[TopicFollowType]]

): PersonalizedContextEligibility = {

val scopedStats = statsReceiver.scope("checkPersonalizedContextsEligibility")

val isRecentFavInAllowlist = allowListWithTopicFollowType

.contains(AnnotationRuleProvider.recentFavTopicId)

val isRecentFavEligible =

isRecentFavInAllowlist && params(TopicSocialProofParams.EnableRecentEngagementsTopic)

if (isRecentFavEligible)

scopedStats.counter("isRecentFavEligible").incr()

val isRecentRetweetInAllowlist = allowListWithTopicFollowType

.contains(AnnotationRuleProvider.recentRetweetTopicId)

val isRecentRetweetEligible =

isRecentRetweetInAllowlist && params(TopicSocialProofParams.EnableRecentEngagementsTopic)

if (isRecentRetweetEligible)

scopedStats.counter("isRecentRetweetEligible").incr()

val isYMLInAllowlist = allowListWithTopicFollowType

.contains(AnnotationRuleProvider.youMightLikeTopicId)

val isYMLEligible =

isYMLInAllowlist && params(TopicSocialProofParams.EnableYouMightLikeTopic)

if (isYMLEligible)

scopedStats.counter("isYMLEligible").incr()

PersonalizedContextEligibility(isRecentFavEligible, isRecentRetweetEligible, isYMLEligible)

}

private[handlers] def filterPersonalizedContexts(

socialProofs: Map[TweetId, Seq[TopicWithScore]],

tweetInfoMap: Map[TweetId, Option[TspTweetInfo]],

params: Params

): Map[TweetId, Seq[TopicWithScore]] = {

val filters: Seq[(Option[TspTweetInfo], Params) => Boolean] = Seq(

healthSignalsFilter,

tweetLanguageFilter

)

applyFilters(socialProofs, tweetInfoMap, params, filters)

}

/\*\* \*

\* filter tweets with None tweetInfo and undefined language

\*/

private def keepTweetsWithTweetInfoAndLanguage(

tweetInfoMapFut: Future[Map[TweetId, Option[TspTweetInfo]]],

displayLocation: String

): Future[Map[TweetId, Option[TspTweetInfo]]] = {

val scopedStats = statsReceiver.scope(displayLocation)

tweetInfoMapFut.map { tweetInfoMap =>

val filteredTweetInfoMap = tweetInfoMap.filter {

case (\_, optTweetInfo: Option[TspTweetInfo]) =>

if (optTweetInfo.isEmpty) {

scopedStats.counter("undefinedTweetInfoCount").incr()

}

optTweetInfo.exists { tweetInfo: TspTweetInfo =>

{

if (tweetInfo.language.isEmpty) {

scopedStats.counter("undefinedLanguageCount").incr()

}

tweetInfo.language.isDefined

}

}

}

val undefinedTweetInfoOrLangCount = tweetInfoMap.size - filteredTweetInfoMap.size

scopedStats.counter("undefinedTweetInfoOrLangCount").incr(undefinedTweetInfoOrLangCount)

scopedStats.counter("TweetInfoCount").incr(tweetInfoMap.size)

filteredTweetInfoMap

}

}

/\*\*\*

\* filter tweets with NO evergreen topic social proofs by their health signal scores & tweet languages

\* i.e., tweets that are possible to be converted into Personalized Context topic tweets

\* TBD: whether we are going to apply filters to all topic tweet candidates

\*/

private def applyFilters(

socialProofs: Map[TweetId, Seq[TopicWithScore]],

tweetInfoMap: Map[TweetId, Option[TspTweetInfo]],

params: Params,

filters: Seq[(Option[TspTweetInfo], Params) => Boolean]

): Map[TweetId, Seq[TopicWithScore]] = {

socialProofs.collect {

case (tweetId, socialProofs) if socialProofs.nonEmpty || filters.forall { filter =>

filter(tweetInfoMap.getOrElse(tweetId, None), params)

} =>

tweetId -> socialProofs

}

}

private def healthSignalsFilter(

tweetInfoOpt: Option[TspTweetInfo],

params: Params

): Boolean = {

!params(

TopicSocialProofParams.EnableTopicTweetHealthFilterPersonalizedContexts) || HealthSignalsUtils

.isHealthyTweet(tweetInfoOpt)

}

private def tweetLanguageFilter(

tweetInfoOpt: Option[TspTweetInfo],

params: Params

): Boolean = {

PersonalizedContextTopicsAllowedLanguageSet

.contains(tweetInfoOpt.flatMap(\_.language).getOrElse(LocaleUtil.DefaultLanguage))

}

private[handlers] def backfillPersonalizedContexts(

allowListWithTopicFollowType: Map[SemanticCoreEntityId, Option[TopicFollowType]],

socialProofs: Map[TweetId, Seq[TopicWithScore]],

metricTagsMap: scala.collection.Map[TweetId, scala.collection.Set[MetricTag]],

personalizedContextEligibility: PersonalizedContextEligibility

): Map[TweetId, Seq[TopicWithScore]] = {

val scopedStats = statsReceiver.scope("backfillPersonalizedContexts")

socialProofs.map {

case (tweetId, topicWithScores) =>

if (topicWithScores.nonEmpty) {

tweetId -> Seq.empty

} else {

val metricTagContainsTweetFav = metricTagsMap

.getOrElse(tweetId, Set.empty[MetricTag]).contains(MetricTag.TweetFavorite)

val backfillRecentFav =

personalizedContextEligibility.isRecentFavEligible && metricTagContainsTweetFav

if (metricTagContainsTweetFav)

scopedStats.counter("MetricTag.TweetFavorite").incr()

if (backfillRecentFav)

scopedStats.counter("backfillRecentFav").incr()

val metricTagContainsRetweet = metricTagsMap

.getOrElse(tweetId, Set.empty[MetricTag]).contains(MetricTag.Retweet)

val backfillRecentRetweet =

personalizedContextEligibility.isRecentRetweetEligible && metricTagContainsRetweet

if (metricTagContainsRetweet)

scopedStats.counter("MetricTag.Retweet").incr()

if (backfillRecentRetweet)

scopedStats.counter("backfillRecentRetweet").incr()

val metricTagContainsRecentSearches = metricTagsMap

.getOrElse(tweetId, Set.empty[MetricTag]).contains(

MetricTag.InterestsRankerRecentSearches)

val backfillYML = personalizedContextEligibility.isYMLEligible

if (backfillYML)

scopedStats.counter("backfillYML").incr()

tweetId -> buildBackfillTopics(

allowListWithTopicFollowType,

backfillRecentFav,

backfillRecentRetweet,

backfillYML)

}

}

}

private def buildBackfillTopics(

allowListWithTopicFollowType: Map[SemanticCoreEntityId, Option[TopicFollowType]],

backfillRecentFav: Boolean,

backfillRecentRetweet: Boolean,

backfillYML: Boolean

): Seq[TopicWithScore] = {

Seq(

if (backfillRecentFav) {

Some(

TopicWithScore(

topicId = AnnotationRuleProvider.recentFavTopicId,

score = 1.0,

topicFollowType = allowListWithTopicFollowType

.getOrElse(AnnotationRuleProvider.recentFavTopicId, None)

))

} else { None },

if (backfillRecentRetweet) {

Some(

TopicWithScore(

topicId = AnnotationRuleProvider.recentRetweetTopicId,

score = 1.0,

topicFollowType = allowListWithTopicFollowType

.getOrElse(AnnotationRuleProvider.recentRetweetTopicId, None)

))

} else { None },

if (backfillYML) {

Some(

TopicWithScore(

topicId = AnnotationRuleProvider.youMightLikeTopicId,

score = 1.0,

topicFollowType = allowListWithTopicFollowType

.getOrElse(AnnotationRuleProvider.youMightLikeTopicId, None)

))

} else { None }

).flatten

}

def toReadableStore: ReadableStore[TopicSocialProofRequest, TopicSocialProofResponse] = {

new ReadableStore[TopicSocialProofRequest, TopicSocialProofResponse] {

override def get(k: TopicSocialProofRequest): Future[Option[TopicSocialProofResponse]] = {

val displayLocation = k.displayLocation.toString

loadShedder(displayLocation) {

getTopicSocialProofResponse(k).map(Some(\_))

}.rescue {

case LoadShedder.LoadSheddingException =>

statsReceiver.scope(displayLocation).counter("LoadSheddingException").incr()

Future.None

case \_ =>

statsReceiver.scope(displayLocation).counter("Exception").incr()

Future.None

}

}

}

}

}

object TopicSocialProofHandler {

private val MaxCandidates = 10

// Currently we do hardcode for the language check of PersonalizedContexts Topics

private val PersonalizedContextTopicsAllowedLanguageSet: Set[String] =

Set("pt", "ko", "es", "ja", "tr", "id", "en", "hi", "ar", "fr", "ru")

private val Timeout: Duration = 200.milliseconds

private val TopicSocialProofStoreTimeout: Duration = 40.milliseconds

private val TweetInfoStoreTimeout: Duration = 60.milliseconds

private val DefaultResponse: TopicSocialProofResponse = TopicSocialProofResponse(Map.empty)

case class PersonalizedContextEligibility(

isRecentFavEligible: Boolean,

isRecentRetweetEligible: Boolean,

isYMLEligible: Boolean)

/\*\*

\* Calculate the Topic Scores for each (tweet, topic), filter out topic proofs whose scores do not

\* pass the minimum threshold

\*/

private[handlers] def buildTopicWithValidScore(

topicProof: TopicSocialProof,

tweetEmbeddingType: EmbeddingType,

maybeConsumerEmbeddingType: Option[EmbeddingType],

maybeProducerEmbeddingType: Option[EmbeddingType],

allowListWithTopicFollowType: Map[SemanticCoreEntityId, Option[TopicFollowType]],

simClustersModelVersion: ModelVersion,

minTweetToTopicCosineSimilarityThreshold: Double

): Option[TopicWithScore] = {

val consumerScore = maybeConsumerEmbeddingType

.flatMap { consumerEmbeddingType =>

topicProof.scores.get(

ScoreKey(consumerEmbeddingType, tweetEmbeddingType, simClustersModelVersion))

}.getOrElse(0.0)

val producerScore = maybeProducerEmbeddingType

.flatMap { producerEmbeddingType =>

topicProof.scores.get(

ScoreKey(producerEmbeddingType, tweetEmbeddingType, simClustersModelVersion))

}.getOrElse(0.0)

val combinedScore = consumerScore + producerScore

if (combinedScore > minTweetToTopicCosineSimilarityThreshold || topicProof.ignoreSimClusterFiltering) {

Some(

TopicWithScore(

topicId = topicProof.topicId.entityId,

score = combinedScore,

topicFollowType =

allowListWithTopicFollowType.getOrElse(topicProof.topicId.entityId, None)))

} else {

None

}

}

private[handlers] def buildTopicWithRandomScore(

topicSocialProof: TopicSocialProof,

allowListWithTopicFollowType: Map[SemanticCoreEntityId, Option[TopicFollowType]],

random: Random

): Option[TopicWithScore] = {

Some(

TopicWithScore(

topicId = topicSocialProof.topicId.entityId,

score = random.nextDouble(),

topicFollowType =

allowListWithTopicFollowType.getOrElse(topicSocialProof.topicId.entityId, None)

))

}

/\*\*

\* Filter all the non-qualified Topic Social Proof

\*/

private[handlers] def filterByAllowedList(

topicProofs: Map[TweetId, Seq[TopicSocialProof]],

setting: TopicListingSetting,

allowList: Set[SemanticCoreEntityId]

): Map[TweetId, Seq[TopicSocialProof]] = {

setting match {

case TopicListingSetting.All =>

// Return all the topics

topicProofs

case \_ =>

topicProofs.mapValues(

\_.filter(topicProof => allowList.contains(topicProof.topicId.entityId)))

}

}

}