package com.twitter.frigate.pushservice.target

import com.twitter.abdecider.LoggingABDecider

import com.twitter.conversions.DurationOps.\_

import com.twitter.decider.Decider

import com.twitter.discovery.common.configapi.ConfigParamsBuilder

import com.twitter.discovery.common.configapi.ExperimentOverride

import com.twitter.featureswitches.Recipient

import com.twitter.finagle.stats.StatsReceiver

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

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

import com.twitter.frigate.common.logger.MRLogger

import com.twitter.frigate.common.store.FeedbackRequest

import com.twitter.frigate.common.store.PushRecItemsKey

import com.twitter.frigate.common.store.deviceinfo.DeviceInfo

import com.twitter.frigate.common.store.interests.UserId

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

import com.twitter.frigate.data\_pipeline.features\_common.MrRequestContextForFeatureStore

import com.twitter.frigate.data\_pipeline.thriftscala.UserHistoryValue

import com.twitter.frigate.dau\_model.thriftscala.DauProbability

import com.twitter.frigate.pushcap.thriftscala.PushcapInfo

import com.twitter.frigate.pushcap.thriftscala.PushcapUserHistory

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

import com.twitter.frigate.pushservice.ml.HydrationContextBuilder

import com.twitter.frigate.pushservice.ml.PushMLModelScorer

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

import com.twitter.frigate.pushservice.params.PushParams

import com.twitter.frigate.pushservice.store.LabeledPushRecsStoreKey

import com.twitter.frigate.pushservice.store.OnlineUserHistoryKey

import com.twitter.frigate.pushservice.util.NsfwInfo

import com.twitter.frigate.pushservice.util.NsfwPersonalizationUtil

import com.twitter.frigate.pushservice.util.PushAppPermissionUtil

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

import com.twitter.frigate.pushservice.thriftscala.PushContext

import com.twitter.frigate.pushservice.thriftscala.RequestSource

import com.twitter.frigate.thriftscala.SecondaryAccountsByUserState

import com.twitter.frigate.thriftscala.UserForPushTargeting

import com.twitter.frigate.user\_states.thriftscala.MRUserHmmState

import com.twitter.frigate.user\_states.thriftscala.{UserState => MrUserState}

import com.twitter.frontpage.stream.util.SnowflakeUtil

import com.twitter.geoduck.common.thriftscala.Place

import com.twitter.geoduck.service.thriftscala.LocationResponse

import com.twitter.gizmoduck.thriftscala.User

import com.twitter.hermit.model.user\_state.UserState

import com.twitter.hermit.model.user\_state.UserState.UserState

import com.twitter.hermit.stp.thriftscala.STPResult

import com.twitter.ibis.thriftscala.ContentRecData

import com.twitter.interests.thriftscala.InterestId

import com.twitter.notificationservice.feedback.thriftscala.FeedbackInteraction

import com.twitter.notificationservice.genericfeedbackstore.FeedbackPromptValue

import com.twitter.notificationservice.genericfeedbackstore.GenericFeedbackStore

import com.twitter.notificationservice.genericfeedbackstore.GenericFeedbackStoreException

import com.twitter.notificationservice.model.service.DismissMenuFeedbackAction

import com.twitter.notificationservice.scribe.manhattan.GenericNotificationsFeedbackRequest

import com.twitter.notificationservice.thriftscala.CaretFeedbackDetails

import com.twitter.nrel.heavyranker.FeatureHydrator

import com.twitter.nrel.hydration.push.HydrationContext

import com.twitter.permissions\_storage.thriftscala.AppPermission

import com.twitter.rux.common.strato.thriftscala.UserTargetingProperty

import com.twitter.scio.nsfw\_user\_segmentation.thriftscala.NSFWUserSegmentation

import com.twitter.service.metastore.gen.thriftscala.Location

import com.twitter.service.metastore.gen.thriftscala.UserLanguages

import com.twitter.stitch.Stitch

import com.twitter.stitch.tweetypie.TweetyPie.TweetyPieResult

import com.twitter.storehaus.ReadableStore

import com.twitter.timelines.configapi

import com.twitter.timelines.real\_graph.thriftscala.{RealGraphFeatures => RealGraphFeaturesUnion}

import com.twitter.timelines.real\_graph.v1.thriftscala.RealGraphFeatures

import com.twitter.ubs.thriftscala.SellerApplicationState

import com.twitter.ubs.thriftscala.SellerTrack

import com.twitter.user\_session\_store.thriftscala.UserSession

import com.twitter.util.Duration

import com.twitter.util.Future

import com.twitter.util.Time

import com.twitter.wtf.scalding.common.thriftscala.UserFeatures

case class PushTargetUserBuilder(

historyStore: PushServiceHistoryStore,

emailHistoryStore: PushServiceHistoryStore,

labeledPushRecsStore: ReadableStore[LabeledPushRecsStoreKey, UserHistoryValue],

onlineUserHistoryStore: ReadableStore[OnlineUserHistoryKey, UserHistoryValue],

pushRecItemsStore: ReadableStore[PushRecItemsKey, RecItems],

userStore: ReadableStore[Long, User],

pushInfoStore: ReadableStore[Long, UserForPushTargeting],

userCountryStore: ReadableStore[Long, Location],

userUtcOffsetStore: ReadableStore[Long, Duration],

dauProbabilityStore: ReadableStore[Long, DauProbability],

nsfwConsumerStore: ReadableStore[Long, NSFWUserSegmentation],

userFeatureStore: ReadableStore[Long, UserFeatures],

userTargetingPropertyStore: ReadableStore[Long, UserTargetingProperty],

mrUserStateStore: ReadableStore[Long, MRUserHmmState],

tweetImpressionStore: ReadableStore[Long, Seq[Long]],

ntabCaretFeedbackStore: ReadableStore[GenericNotificationsFeedbackRequest, Seq[

CaretFeedbackDetails

]],

genericFeedbackStore: ReadableStore[FeedbackRequest, Seq[FeedbackPromptValue]],

genericNotificationFeedbackStore: GenericFeedbackStore,

timelinesUserSessionStore: ReadableStore[Long, UserSession],

cachedTweetyPieStore: ReadableStore[Long, TweetyPieResult],

strongTiesStore: ReadableStore[Long, STPResult],

userHTLLastVisitStore: ReadableStore[Long, Seq[Long]],

userLanguagesStore: ReadableStore[Long, UserLanguages],

inputDecider: Decider,

inputAbDecider: LoggingABDecider,

realGraphScoresTop500InStore: ReadableStore[Long, Map[Long, Double]],

recentFollowsStore: ReadableStore[Long, Seq[Long]],

resurrectedUserStore: ReadableStore[Long, String],

configParamsBuilder: ConfigParamsBuilder,

optOutUserInterestsStore: ReadableStore[UserId, Seq[InterestId]],

deviceInfoStore: ReadableStore[Long, DeviceInfo],

pushcapDynamicPredictionStore: ReadableStore[Long, PushcapUserHistory],

appPermissionStore: ReadableStore[(Long, (String, String)), AppPermission],

optoutModelScorer: PushMLModelScorer,

inlineActionHistoryStore: ReadableStore[Long, Seq[(Long, String)]],

featureHydrator: FeatureHydrator,

openAppUserStore: ReadableStore[Long, Boolean],

openedPushByHourAggregatedStore: ReadableStore[Long, Map[Int, Int]],

geoduckStoreV2: ReadableStore[Long, LocationResponse],

superFollowEligibilityUserStore: ReadableStore[Long, Boolean],

superFollowApplicationStatusStore: ReadableStore[(Long, SellerTrack), SellerApplicationState]

)(

globalStatsReceiver: StatsReceiver) {

implicit val statsReceiver: StatsReceiver = globalStatsReceiver

private val log = MRLogger("PushTargetUserBuilder")

private val recentFollowscounter = statsReceiver.counter("query\_recent\_follows")

private val isModelTrainingDataCounter =

statsReceiver.scope("TargetUserBuilder").counter("is\_model\_training")

private val feedbackStoreGenerationErr = statsReceiver.counter("feedback\_store\_generation\_error")

private val newSignUpUserStats = statsReceiver.counter("new\_signup\_user")

private val pushcapSelectionStat = statsReceiver.scope("pushcap\_modeling")

private val dormantUserCount = statsReceiver.counter("dormant\_user\_counter")

private val optoutModelStat = statsReceiver.scope("optout\_modeling")

private val placeFoundStat = statsReceiver.scope("geoduck\_v2").stat("places\_found")

private val placesNotFound = statsReceiver.scope("geoduck\_v2").counter("places\_not\_found")

// Email history store stats

private val emailHistoryStats = statsReceiver.scope("email\_tweet\_history")

private val emptyEmailHistoryCounter = emailHistoryStats.counter("empty")

private val nonEmptyEmailHistoryCounter = emailHistoryStats.counter("non\_empty")

private val MagicRecsCategory = "MagicRecs"

private val MomentsViaMagicRecsCategory = "MomentsViaMagicRecs"

private val MomentsCategory = "Moments"

def buildTarget(

userId: Long,

inputPushContext: Option[PushContext],

forcedFeatureValues: Option[Map[String, configapi.FeatureValue]] = None

): Future[Target] = {

val historyStoreKeyContext = HistoryStoreKeyContext(

userId,

inputPushContext.flatMap(\_.useMemcacheForHistory).getOrElse(false)

)

Future

.join(

userStore.get(userId),

deviceInfoStore.get(userId),

pushInfoStore.get(userId),

historyStore.get(historyStoreKeyContext, Some(30.days)),

emailHistoryStore.get(

HistoryStoreKeyContext(userId, useStoreB = false),

Some(7.days) // we only keep 7 days of email tweet history

)

).flatMap {

case (userOpt, deviceInfoOpt, userForPushTargetingInfoOpt, notifHistory, emailHistory) =>

getCustomFSFields(

userId,

userOpt,

deviceInfoOpt,

userForPushTargetingInfoOpt,

notifHistory,

inputPushContext.flatMap(\_.requestSource)).map { customFSField =>

new Target {

override lazy val stats: StatsReceiver = statsReceiver

override val targetId: Long = userId

override val targetUser: Future[Option[User]] = Future.value(userOpt)

override val isEmailUser: Boolean =

inputPushContext.flatMap(\_.requestSource) match {

case Some(source) if source == RequestSource.Email => true

case \_ => false

}

override val pushContext = inputPushContext

override def globalStats: StatsReceiver = globalStatsReceiver

override lazy val abDecider: ABDeciderWithOverride =

ABDeciderWithOverride(inputAbDecider, ddgOverrideOption)

override lazy val pushRecItems: Future[RecItems] =

pushRecItemsStore

.get(PushRecItemsKey(historyStoreKeyContext, history))

.map(\_.getOrElse(RecItems.empty))

// List of past tweet candidates sent in the past through email with timestamp

override lazy val emailRecItems: Future[Seq[(Time, Long)]] = {

Future.value {

emailHistory.sortedEmailHistory.flatMap {

case (timeStamp, notification) =>

notification.contentRecsNotification

.map { notification =>

notification.recommendations.contentRecCollections.flatMap {

contentRecs =>

contentRecs.contentRecModules.flatMap { contentRecModule =>

contentRecModule.recData match {

case ContentRecData.TweetRec(tweetRec) =>

nonEmptyEmailHistoryCounter.incr()

Seq(tweetRec.tweetId)

case \_ =>

emptyEmailHistoryCounter.incr()

Nil

}

}

}

}.getOrElse {

emptyEmailHistoryCounter.incr()

Nil

}.map(timeStamp -> \_)

}

}

}

override lazy val history: Future[History] = Future.value(notifHistory)

override lazy val pushTargeting: Future[Option[UserForPushTargeting]] =

Future.value(userForPushTargetingInfoOpt)

override lazy val decider: Decider = inputDecider

override lazy val location: Future[Option[Location]] =

userCountryStore.get(userId)

override lazy val deviceInfo: Future[Option[DeviceInfo]] =

Future.value(deviceInfoOpt)

override lazy val targetLanguage: Future[Option[String]] = targetUser map { userOpt =>

userOpt.flatMap(\_.account.map(\_.language))

}

override lazy val targetAgeInYears: Future[Option[Int]] =

Future.value(customFSField.userAge)

override lazy val metastoreLanguages: Future[Option[UserLanguages]] =

userLanguagesStore.get(targetId)

override lazy val utcOffset: Future[Option[Duration]] =

userUtcOffsetStore.get(targetId)

override lazy val userFeatures: Future[Option[UserFeatures]] =

userFeatureStore.get(targetId)

override lazy val targetUserState: Future[Option[UserState]] =

Future.value(

customFSField.userState

.flatMap(userState => UserState.valueOf(userState)))

override lazy val targetMrUserState: Future[Option[MrUserState]] =

Future.value(

customFSField.mrUserState

.flatMap(mrUserState => MrUserState.valueOf(mrUserState)))

override lazy val accountStateWithDeviceInfo: Future[

Option[SecondaryAccountsByUserState]

] = Future.None

override lazy val dauProbability: Future[Option[DauProbability]] = {

dauProbabilityStore.get(targetId)

}

override lazy val labeledPushRecsHydrated: Future[Option[UserHistoryValue]] =

labeledPushRecsStore.get(LabeledPushRecsStoreKey(this, historyStoreKeyContext))

override lazy val onlineLabeledPushRecs: Future[Option[UserHistoryValue]] =

labeledPushRecsHydrated.flatMap { labeledPushRecs =>

history.flatMap { history =>

onlineUserHistoryStore.get(

OnlineUserHistoryKey(targetId, labeledPushRecs, Some(history))

)

}

}

override lazy val tweetImpressionResults: Future[Seq[Long]] =

tweetImpressionStore.get(targetId).map {

case Some(impressionList) =>

impressionList

case \_ => Nil

}

override lazy val realGraphFeatures: Future[Option[RealGraphFeatures]] =

timelinesUserSessionStore.get(targetId).map { userSessionOpt =>

userSessionOpt.flatMap { userSession =>

userSession.realGraphFeatures.collect {

case RealGraphFeaturesUnion.V1(rGFeatures) =>

rGFeatures

}

}

}

override lazy val stpResult: Future[Option[STPResult]] =

strongTiesStore.get(targetId)

override lazy val lastHTLVisitTimestamp: Future[Option[Long]] =

userHTLLastVisitStore.get(targetId).map {

case Some(lastVisitTimestamps) if lastVisitTimestamps.nonEmpty =>

Some(lastVisitTimestamps.max)

case \_ => None

}

override lazy val caretFeedbacks: Future[Option[Seq[CaretFeedbackDetails]]] = {

val scribeHistoryLookbackPeriod = 365.days

val now = Time.now

val request = GenericNotificationsFeedbackRequest(

userId = targetId,

eventStartTimestamp = now - scribeHistoryLookbackPeriod,

eventEndTimestamp = now,

filterCategory =

Some(Set(MagicRecsCategory, MomentsViaMagicRecsCategory, MomentsCategory)),

filterFeedbackActionText =

Some(Set(DismissMenuFeedbackAction.FeedbackActionTextSeeLessOften))

)

ntabCaretFeedbackStore.get(request)

}

override lazy val notificationFeedbacks: Future[

Option[Seq[FeedbackPromptValue]]

] = {

val scribeHistoryLookbackPeriod = 30.days

val now = Time.now

val request = FeedbackRequest(

userId = targetId,

oldestTimestamp = scribeHistoryLookbackPeriod.ago,

newestTimestamp = Time.now,

feedbackInteraction = FeedbackInteraction.Feedback

)

genericFeedbackStore.get(request)

}

// DEPRECATED: Use notificationFeedbacks instead.

// This method will increase latency dramatically.

override lazy val promptFeedbacks: Stitch[Seq[FeedbackPromptValue]] = {

val scribeHistoryLookbackPeriod = 7.days

genericNotificationFeedbackStore

.getAll(

userId = targetId,

oldestTimestamp = scribeHistoryLookbackPeriod.ago,

newestTimestamp = Time.now,

feedbackInteraction = FeedbackInteraction.Feedback

).handle {

case \_: GenericFeedbackStoreException => {

feedbackStoreGenerationErr.incr()

Seq.empty[FeedbackPromptValue]

}

}

}

override lazy val optOutUserInterests: Future[Option[Seq[InterestId]]] = {

optOutUserInterestsStore.get(targetId)

}

private val experimentOverride = ddgOverrideOption.map {

case DDGOverride(Some(exp), Some(bucket)) =>

Set(ExperimentOverride(exp, bucket))

case \_ => Set.empty[ExperimentOverride]

}

override val signupCountryCode =

Future.value(userOpt.flatMap(\_.safety.flatMap(\_.signupCountryCode)))

override lazy val params: configapi.Params = {

val fsRecipient = Recipient(

userId = Some(targetId),

userRoles = userOpt.flatMap(\_.roles.map(\_.roles.toSet)),

clientApplicationId = deviceInfoOpt.flatMap(\_.guessedPrimaryClientAppId),

userAgent = deviceInfoOpt.flatMap(\_.guessedPrimaryDeviceUserAgent),

countryCode =

userOpt.flatMap(\_.account.flatMap(\_.countryCode.map(\_.toUpperCase))),

customFields = Some(customFSField.fsMap),

signupCountryCode =

userOpt.flatMap(\_.safety.flatMap(\_.signupCountryCode.map(\_.toUpperCase))),

languageCode = deviceInfoOpt.flatMap {

\_.deviceLanguages.flatMap(IbisAppPushDeviceSettingsUtil.inferredDeviceLanguage)

}

)

configParamsBuilder.build(

userId = Some(targetId),

experimentOverrides = experimentOverride,

featureRecipient = Some(fsRecipient),

forcedFeatureValues = forcedFeatureValues.getOrElse(Map.empty),

)

}

override lazy val mrRequestContextForFeatureStore =

MrRequestContextForFeatureStore(targetId, params, isModelTrainingData)

override lazy val dynamicPushcap: Future[Option[PushcapInfo]] = {

// Get the pushcap from the pushcap model prediction store

if (params(PushParams.EnableModelBasedPushcapAssignments)) {

val originalPushcapInfoFut =

PushCapUtil.getPushcapFromUserHistory(

userId,

pushcapDynamicPredictionStore,

params(FeatureSwitchParams.PushcapModelType),

params(FeatureSwitchParams.PushcapModelPredictionVersion),

pushcapSelectionStat

)

// Modify the push cap info if there is a restricted min value for predicted push caps.

val restrictedPushcap = params(PushFeatureSwitchParams.RestrictedMinModelPushcap)

originalPushcapInfoFut.map {

case Some(originalPushcapInfo) =>

Some(

getMinimumRestrictedPushcapInfo(

restrictedPushcap,

originalPushcapInfo,

pushcapSelectionStat))

case \_ => None

}

} else Future.value(None)

}

override lazy val targetHydrationContext: Future[HydrationContext] =

HydrationContextBuilder.build(this)

override lazy val featureMap: Future[FeatureMap] =

targetHydrationContext.flatMap { hydrationContext =>

featureHydrator.hydrateTarget(

hydrationContext,

this.params,

this.mrRequestContextForFeatureStore)

}

override lazy val globalOptoutProbabilities: Seq[Future[Option[Double]]] = {

params(PushFeatureSwitchParams.GlobalOptoutModelParam).map { model\_id =>

optoutModelScorer

.singlePredictionForTargetLevel(model\_id, targetId, featureMap)

}

}

override lazy val bucketOptoutProbability: Future[Option[Double]] = {

Future

.collect(globalOptoutProbabilities).map {

\_.zip(params(PushFeatureSwitchParams.GlobalOptoutThresholdParam))

.exists {

case (Some(score), threshold) => score >= threshold

case \_ => false

}

}.flatMap {

case true =>

optoutModelScorer.singlePredictionForTargetLevel(

params(PushFeatureSwitchParams.BucketOptoutModelParam),

targetId,

featureMap)

case \_ => Future.None

}

}

override lazy val optoutAdjustedPushcap: Future[Option[Short]] = {

if (params(PushFeatureSwitchParams.EnableOptoutAdjustedPushcap)) {

bucketOptoutProbability.map {

case Some(score) =>

val idx = params(PushFeatureSwitchParams.BucketOptoutSlotThresholdParam)

.indexWhere(score <= \_)

if (idx >= 0) {

val pushcap =

params(PushFeatureSwitchParams.BucketOptoutSlotPushcapParam)(idx).toShort

optoutModelStat.scope("adjusted\_pushcap").counter(f"$pushcap").incr()

if (pushcap >= 0) Some(pushcap)

else None

} else None

case \_ => None

}

} else Future.None

}

override lazy val seedsWithWeight: Future[Option[Map[Long, Double]]] = {

Future

.join(

realGraphScoresTop500InStore.get(userId),

targetUserState,

targetUser

)

.flatMap {

case (seedSetOpt, userState, gizmoduckUser) =>

val seedSet = seedSetOpt.getOrElse(Map.empty[Long, Double])

//If new sign\_up or New user, combine recent\_follows with real graph seedset

val isNewUserEnabled = {

val isNewerThan7days = customFSField.daysSinceSignup <= 7

val isNewUserState = userState.contains(UserState.New)

isNewUserState || isNewSignup || isNewerThan7days

}

val nonSeedSetFollowsFut = gizmoduckUser match {

case Some(user) if isNewUserEnabled =>

recentFollowscounter.incr()

recentFollowsStore.get(user.id)

case Some(user) if this.isModelTrainingData =>

recentFollowscounter.incr()

isModelTrainingDataCounter.incr()

recentFollowsStore.get(user.id)

case \_ => Future.None

}

nonSeedSetFollowsFut.map { nonSeedSetFollows =>

Some(

SeedsetUtil.combineRecentFollowsWithWeightedSeedset(

seedSet,

nonSeedSetFollows.getOrElse(Nil)

)

)

}

}

}

override def magicFanoutReasonHistory30Days: Future[MagicFanoutReasonHistory] =

history.map(history => MagicFanoutReasonHistory(history))

override val isNewSignup: Boolean =

pushContext.flatMap(\_.isFromNewUserLoopProcessor).getOrElse(false)

override lazy val resurrectionDate: Future[Option[String]] =

Future.value(customFSField.reactivationDate)

override lazy val isResurrectedUser: Boolean =

customFSField.daysSinceReactivation.isDefined

override lazy val timeSinceResurrection: Option[Duration] =

customFSField.daysSinceReactivation.map(Duration.fromDays)

override lazy val appPermissions: Future[Option[AppPermission]] =

PushAppPermissionUtil.getAppPermission(

userId,

PushAppPermissionUtil.AddressBookPermissionKey,

deviceInfo,

appPermissionStore)

override lazy val inlineActionHistory: Future[Seq[(Long, String)]] = {

inlineActionHistoryStore

.get(userId).map {

case Some(sortedInlineActionHistory) => sortedInlineActionHistory

case \_ => Seq.empty

}

}

lazy val isOpenAppExperimentUser: Future[Boolean] =

openAppUserStore.get(userId).map(\_.contains(true))

override lazy val openedPushByHourAggregated: Future[Option[Map[Int, Int]]] =

openedPushByHourAggregatedStore.get(userId)

override lazy val places: Future[Seq[Place]] = {

geoduckStoreV2

.get(targetId)

.map(\_.flatMap(\_.places))

.map {

case Some(placeSeq) if placeSeq.nonEmpty =>

placeFoundStat.add(placeSeq.size)

placeSeq

case \_ =>

placesNotFound.incr()

Seq.empty

}

}

override val isBlueVerified: Future[Option[Boolean]] =

Future.value(userOpt.flatMap(\_.safety.flatMap(\_.isBlueVerified)))

override val isVerified: Future[Option[Boolean]] =

Future.value(userOpt.flatMap(\_.safety.map(\_.verified)))

override lazy val isSuperFollowCreator: Future[Option[Boolean]] =

superFollowEligibilityUserStore.get(targetId)

}

}

}

}

/\*\*

\* Provide general way to add needed FS for target user, and package them in CustomFSFields.

\* Custom Fields is a powerful feature that allows Feature Switch library users to define and

\* match against any arbitrary fields.

\*\*/

private def getCustomFSFields(

userId: Long,

userOpt: Option[User],

deviceInfo: Option[DeviceInfo],

userForPushTargetingInfo: Option[UserForPushTargeting],

notifHistory: History,

requestSource: Option[RequestSource]

): Future[CustomFSFields] = {

val reactivationDateFutOpt: Future[Option[String]] = resurrectedUserStore.get(userId)

val reactivationTimeFutOpt: Future[Option[Time]] =

reactivationDateFutOpt.map(\_.map(dateStr => DateUtil.dateStrToTime(dateStr)))

val isReactivatedUserFut: Future[Boolean] = reactivationTimeFutOpt.map { timeOpt =>

timeOpt

.exists { time => Time.now - time < 30.days }

}

val daysSinceReactivationFut: Future[Option[Int]] =

reactivationTimeFutOpt.map(\_.map(time => Time.now.since(time).inDays))

val daysSinceSignup: Int = (Time.now - SnowflakeUtil.timeFromId(userId)).inDays

if (daysSinceSignup < 14) newSignUpUserStats.incr()

val targetAgeInYears = userOpt.flatMap(\_.extendedProfile.flatMap(\_.ageInYears))

val lastLoginFut: Future[Option[Long]] =

userHTLLastVisitStore.get(userId).map {

case Some(lastHTLVisitTimes) =>

val latestHTLVisitTime = lastHTLVisitTimes.max

userForPushTargetingInfo.flatMap(

\_.lastActiveOnAppTimestamp

.map(\_.max(latestHTLVisitTime)).orElse(Some(latestHTLVisitTime)))

case None =>

userForPushTargetingInfo.flatMap(\_.lastActiveOnAppTimestamp)

}

val daysSinceLoginFut = lastLoginFut.map {

\_.map { lastLoginTimestamp =>

val timeSinceLogin = Time.now - Time.fromMilliseconds(lastLoginTimestamp)

if (timeSinceLogin.inDays > 21) {

dormantUserCount.incr()

}

timeSinceLogin.inDays

}

}

/\* Could add more custom FS here \*/

val userNSFWInfoFut: Future[Option[NsfwInfo]] =

nsfwConsumerStore

.get(userId).map(\_.map(nsfwUserSegmentation => NsfwInfo(nsfwUserSegmentation)))

val userStateFut: Future[Option[String]] = userFeatureStore.get(userId).map { userFeaturesOpt =>

userFeaturesOpt.flatMap { uFeats =>

uFeats.userState.map(uState => uState.name)

}

}

val mrUserStateFut: Future[Option[String]] =

mrUserStateStore.get(userId).map { mrUserStateOpt =>

mrUserStateOpt.flatMap { mrUserState =>

mrUserState.userState.map(\_.name)

}

}

Future

.join(

reactivationDateFutOpt,

isReactivatedUserFut,

userStateFut,

mrUserStateFut,

daysSinceLoginFut,

daysSinceReactivationFut,

userNSFWInfoFut

).map {

case (

reactivationDate,

isReactivatedUser,

userState,

mrUserState,

daysSinceLogin,

daysSinceReactivation,

userNSFWInfo) =>

val numDaysReceivedPushInLast30Days: Int =

notifHistory.history.keys.map(\_.inDays).toSet.size

NsfwPersonalizationUtil.computeNsfwUserStats(userNSFWInfo)

CustomFSFields(

isReactivatedUser,

daysSinceSignup,

numDaysReceivedPushInLast30Days,

daysSinceLogin,

daysSinceReactivation,

userOpt,

userState,

mrUserState,

reactivationDate,

requestSource.map(\_.name),

targetAgeInYears,

userNSFWInfo,

deviceInfo

)

}

}

}