package com.twitter.simclusters\_v2.scalding.embedding.common

import com.twitter.algebird.Aggregator

import com.twitter.common.text.language.LocaleUtil

import com.twitter.escherbird.common.thriftscala.Locale

import com.twitter.escherbird.common.thriftscala.LocalizedUser

import com.twitter.escherbird.metadata.thriftscala.FullMetadata

import com.twitter.escherbird.scalding.source.FullMetadataSource

import com.twitter.escherbird.scalding.source.utt.UttSourceScalaDataset

import com.twitter.escherbird.utt.strato.thriftscala.SnapshotType

import com.twitter.escherbird.utt.thriftscala.UttEntityRecord

import com.twitter.interests\_ds.jobs.interests\_service.UserTopicRelationSnapshotScalaDataset

import com.twitter.interests.thriftscala.InterestRelationType

import com.twitter.interests.thriftscala.UserInterestsRelationSnapshot

import com.twitter.penguin.scalding.datasets.PenguinUserLanguagesScalaDataset

import com.twitter.scalding.DateOps

import com.twitter.scalding.DateRange

import com.twitter.scalding.Days

import com.twitter.scalding.Stat

import com.twitter.scalding.TypedPipe

import com.twitter.scalding.UniqueID

import com.twitter.scalding.ValuePipe

import com.twitter.scalding\_internal.dalv2.DAL

import com.twitter.scalding\_internal.dalv2.remote\_access.ExplicitLocation

import com.twitter.scalding\_internal.dalv2.remote\_access.AllowCrossClusterSameDC

import com.twitter.scalding\_internal.dalv2.remote\_access.ProcAtla

import com.twitter.scalding\_internal.multiformat.format.keyval.KeyVal

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

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

import com.twitter.simclusters\_v2.hdfs\_sources.SimclustersV2InterestedIn20M145KUpdatedScalaDataset

import com.twitter.simclusters\_v2.hdfs\_sources.UserUserFavGraphScalaDataset

import com.twitter.scalding\_internal.dalv2.remote\_access.AllowCrossDC

import com.twitter.common\_header.thriftscala.CommonHeader

import com.twitter.common\_header.thriftscala.IdType

import com.twitter.common\_header.thriftscala.VersionedCommonHeader

import flockdb\_tools.datasets.flock.FlockBlocksEdgesScalaDataset

import flockdb\_tools.datasets.flock.FlockFollowsEdgesScalaDataset

import flockdb\_tools.datasets.flock.FlockReportAsAbuseEdgesScalaDataset

import flockdb\_tools.datasets.flock.FlockReportAsSpamEdgesScalaDataset

import twadoop\_config.configuration.log\_categories.group.search.AdaptiveSearchScalaDataset

import com.twitter.search.adaptive.scribing.thriftscala.AdaptiveSearchScribeLog

import twadoop\_config.configuration.log\_categories.group.timeline.TimelineServiceFavoritesScalaDataset

import tweetsource.common.UnhydratedFlatScalaDataset

import com.twitter.frigate.data\_pipeline.magicrecs.magicrecs\_notifications\_lite.thriftscala.MagicRecsNotificationLite

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

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

import com.twitter.timelineservice.thriftscala.ContextualizedFavoriteEvent

import com.twitter.timelineservice.thriftscala.FavoriteEventUnion

import com.twitter.tweetsource.common.thriftscala.UnhydratedFlatTweet

import com.twitter.usersource.snapshot.flat.UsersourceFlatScalaDataset

import com.twitter.wtf.entity\_real\_graph.scalding.common.DatasetConstants

import com.twitter.wtf.entity\_real\_graph.scalding.common.SemanticCoreFilters

import com.twitter.wtf.scalding.client\_event\_processing.thriftscala.InteractionDetails

import com.twitter.wtf.scalding.client\_event\_processing.thriftscala.InteractionType

import com.twitter.wtf.scalding.client\_event\_processing.thriftscala.TweetImpressionDetails

import com.twitter.frigate.data\_pipeline.scalding.magicrecs.magicrecs\_notification\_lite.MagicrecsNotificationLite1DayLagScalaDataset

import com.twitter.iesource.thriftscala.InteractionEvent

import com.twitter.iesource.thriftscala.InteractionTargetType

import com.twitter.wtf.scalding.jobs.client\_event\_processing.UserInteractionScalaDataset

import java.util.TimeZone

import com.twitter.interests\_ds.jobs.interests\_service.UserInterestRelationSnapshotScalaDataset

import com.twitter.simclusters\_v2.scalding.embedding.common.EmbeddingUtil.UserId

import com.twitter.scalding.typed.{ValuePipe => TypedValuePipe}

import com.twitter.tweetsource.common.thriftscala.UnhydratedTweet

import tweetsource.common.UnhydratedScalaDataset

object ExternalDataSources {

val UTTDomain = 131L

val usersourceColumns = Set("id", "account\_country\_code", "language")

val ValidFlockEdgeStateId = 0

def getStandardLanguageCode(language: String): Option[String] = {

val locale = LocaleUtil.getLocaleOf(language)

if (locale == LocaleUtil.UNKNOWN) None else Some(locale.getLanguage)

}

// Reads UTT Entity Records (`utt\_source` dataset)

def getUttEntityRecords(implicit timeZone: TimeZone): TypedPipe[UttEntityRecord] = {

DAL

.readMostRecentSnapshotNoOlderThan(UttSourceScalaDataset, Days(14))

.withRemoteReadPolicy(ExplicitLocation(ProcAtla))

.toTypedPipe

}

/\*\*

\* Extracts the KGO seeds from the UTT Entity Records.

\* Uses the most recent "Stable" version by default unless specified otherwise.

\*

\* @param uttVersion UTT Version to use instead of the default value.

\*/

def getLocaleProducerSeedIdsFromUttEntityRecords(

uttVersion: Option[Long] = None

)(

implicit timeZone: TimeZone,

uniqueId: UniqueID

): TypedPipe[((TopicId, Language), Seq[UserId])] = {

val topicLangPairCount = Stat("topic\_lang\_pair\_count\_all")

val topicLangPairCountEmptySeed = Stat("topic\_lang\_pair\_count\_empty\_seed")

val topicLangPairCountLteOneSeed = Stat("topic\_lang\_pair\_count\_lte\_one\_seed")

val topicLangPairCountLteFiveSeeds = Stat("topic\_lang\_pair\_count\_lte\_five\_seeds")

val topicLangPairCountLteTenSeeds = Stat("topic\_lang\_pair\_count\_lte\_ten\_seeds")

val uttEntityRecords: TypedPipe[UttEntityRecord] = getUttEntityRecords

val uttVersionToUse: ValuePipe[Long] = uttVersion match {

case Some(uttVersionValue) =>

TypedValuePipe(uttVersionValue)

case \_ => // find the most recent "stable" version as recommended by the SemanticCore team

uttEntityRecords

.filter(\_.snapshotType.exists(\_ == SnapshotType.Stable))

.map(\_.version)

.distinct

.aggregate(Aggregator.min) // the most recent version is the smallest negative value

}

val uttEntityRecordsSingleVersion: TypedPipe[UttEntityRecord] =

uttEntityRecords

.filterWithValue(uttVersionToUse) {

case (uttEntityRecord: UttEntityRecord, uttVersionOpt: Option[Long]) =>

uttVersionOpt.contains(uttEntityRecord.version)

}

uttEntityRecordsSingleVersion.flatMap { uttEntityRecord: UttEntityRecord =>

val localizedUsers: Seq[LocalizedUser] =

uttEntityRecord.knownForUsers.flatMap(\_.localizedUsers).getOrElse(Nil)

val validLocalizedUsers: Seq[(TopicId, Language, UserId)] =

localizedUsers

.flatMap {

case LocalizedUser(userId: UserId, Some(Locale(Some(language: String), \_)), \_) =>

Some((uttEntityRecord.entityId, language, userId))

case \_ =>

None

}

val localeProducerSeedIds: Seq[((TopicId, Language), Seq[UserId])] = validLocalizedUsers

.groupBy {

case (topicId: TopicId, language: Language, \_) =>

(topicId, language)

}

.mapValues(\_.map(\_.\_3).distinct) // values are distinct producerIds

.toSeq

localeProducerSeedIds.foreach { // stats

case (\_, seedIds: Seq[UserId]) =>

topicLangPairCount.inc()

if (seedIds.isEmpty) topicLangPairCountEmptySeed.inc()

if (seedIds.length <= 1) topicLangPairCountLteOneSeed.inc()

if (seedIds.length <= 5) topicLangPairCountLteFiveSeeds.inc()

if (seedIds.length <= 10) topicLangPairCountLteTenSeeds.inc()

}

localeProducerSeedIds

}.forceToDisk

}

def uttEntitiesSource(

customFullMetadataSource: Option[TypedPipe[FullMetadata]] = None

)(

implicit dateRange: DateRange,

timeZone: TimeZone

): TypedPipe[Long] = {

customFullMetadataSource

.getOrElse(fullMetadataSource)

.flatMap {

case fullMetadata if fullMetadata.domainId == UTTDomain =>

for {

basicMetadata <- fullMetadata.basicMetadata

indexableFields <- basicMetadata.indexableFields

tags <- indexableFields.tags

if !SemanticCoreFilters.shouldFilterByTags(tags.toSet, DatasetConstants.stopTags)

} yield {

fullMetadata.entityId

}

case \_ => None

}

}

// Get followable topics from Escherbird

def uttFollowableEntitiesSource(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): TypedPipe[Long] = {

val followableEntityCount = Stat("followable\_entities\_count")

val FollowableTag = "utt:followable\_topic"

fullMetadataSource

.flatMap {

case fullMetadata if fullMetadata.domainId == UTTDomain =>

for {

basicMetadata <- fullMetadata.basicMetadata

indexableFields <- basicMetadata.indexableFields

tags <- indexableFields.tags

if tags.contains(FollowableTag)

} yield {

followableEntityCount.inc()

fullMetadata.entityId

}

case \_ => None

}

}

def fullMetadataSource(

implicit dateRange: DateRange,

timeZone: TimeZone

): TypedPipe[FullMetadata] = {

TypedPipe

.from(

new FullMetadataSource(s"/atla/proc/${FullMetadataSource.DefaultHdfsPath}")()(

dateRange.embiggen(Days(7))))

}

def userSource(implicit timeZone: TimeZone): TypedPipe[(UserId, (Country, Language))] =

DAL

.readMostRecentSnapshotNoOlderThan(UsersourceFlatScalaDataset, Days(7))

.withRemoteReadPolicy(ExplicitLocation(ProcAtla))

.withColumns(usersourceColumns)

.toTypedPipe.flatMap { flatUser =>

for {

userId <- flatUser.id

country <- flatUser.accountCountryCode

language <- flatUser.language

standardLang <- getStandardLanguageCode(language)

} yield {

(userId, country.toUpperCase, standardLang)

}

}.distinct

.map { case (user, country, lang) => user -> (country, lang) }

// Build user language source from inferred languages (penguin\_user\_languages dataset)

def inferredUserConsumedLanguageSource(

implicit timeZone: TimeZone

): TypedPipe[(UserId, Seq[(Language, Double)])] = {

DAL

.readMostRecentSnapshotNoOlderThan(PenguinUserLanguagesScalaDataset, Days(7))

.withRemoteReadPolicy(ExplicitLocation(ProcAtla))

.toTypedPipe

.map { kv =>

val consumed = kv.value.consumed

.collect {

case scoredString if scoredString.weight > 0.001 => //throw away 5% outliers

(getStandardLanguageCode(scoredString.item), scoredString.weight)

}.collect {

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

}

(kv.key, consumed)

}

}

def inferredUserProducedLanguageSource(

implicit timeZone: TimeZone

): TypedPipe[(UserId, Seq[(Language, Double)])] = {

DAL

.readMostRecentSnapshotNoOlderThan(PenguinUserLanguagesScalaDataset, Days(7))

.withRemoteReadPolicy(ExplicitLocation(ProcAtla))

.toTypedPipe

.map { kv =>

val produced = kv.value.produced

.collect {

case scoredString if scoredString.weight > 0.15 => //throw away 5% outliers

(getStandardLanguageCode(scoredString.item), scoredString.weight)

}.collect {

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

}

(kv.key, produced)

}

}

def simClustersInterestInSource(

implicit dateRange: DateRange,

timeZone: TimeZone

): TypedPipe[KeyVal[UserId, ClustersUserIsInterestedIn]] = {

DAL

.readMostRecentSnapshotNoOlderThan(

SimclustersV2InterestedIn20M145KUpdatedScalaDataset,

Days(30))

.withRemoteReadPolicy(ExplicitLocation(ProcAtla))

.toTypedPipe

}

def simClustersInterestInLogFavSource(

minLogFavScore: Double

)(

implicit dateRange: DateRange,

timeZone: TimeZone

): TypedPipe[(UserId, Map[ClusterId, Double])] = {

simClustersInterestInSource.map {

case KeyVal(userId, clustersUserIsInterestedIn) =>

userId -> clustersUserIsInterestedIn.clusterIdToScores

.map {

case (clusterId, scores) =>

clusterId -> scores.logFavScore.getOrElse(0.0)

}

.filter(\_.\_2 > minLogFavScore)

.toMap

}

}

def topicFollowGraphSource(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): TypedPipe[(TopicId, UserId)] = {

val userTopicFollowCount = Stat("user\_topic\_follow\_count")

DAL

.readMostRecentSnapshotNoOlderThan(UserTopicRelationSnapshotScalaDataset, Days(7))

.withRemoteReadPolicy(ExplicitLocation(ProcAtla))

.toTypedPipe

.collect {

case userInterestsRelationSnapshot: UserInterestsRelationSnapshot

if userInterestsRelationSnapshot.interestType == "UTT" &&

userInterestsRelationSnapshot.relation == InterestRelationType.Followed =>

(userInterestsRelationSnapshot.interestId, userInterestsRelationSnapshot.userId)

}

.hashJoin(uttFollowableEntitiesSource.asKeys)

.map {

case (topic, (user, \_)) =>

userTopicFollowCount.inc()

(topic, user)

}

}

def notInterestedTopicsSource(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): TypedPipe[(TopicId, UserId)] = {

val userNotInterestedInTopicsCount = Stat("user\_not\_interested\_in\_topics\_count")

DAL

.readMostRecentSnapshotNoOlderThan(

UserInterestRelationSnapshotScalaDataset,

Days(7)).withRemoteReadPolicy(ExplicitLocation(ProcAtla)).toTypedPipe.collect {

case userInterestsRelationSnapshot: UserInterestsRelationSnapshot

if userInterestsRelationSnapshot.interestType == "UTT" &&

userInterestsRelationSnapshot.relation == InterestRelationType.NotInterested =>

(userInterestsRelationSnapshot.interestId, userInterestsRelationSnapshot.userId)

}

.hashJoin(uttFollowableEntitiesSource.asKeys)

.map {

case (topic, (user, \_)) =>

userNotInterestedInTopicsCount.inc()

(topic, user)

}

}

def tweetSource(

implicit dateRange: DateRange

): TypedPipe[UnhydratedTweet] = {

DAL

.read(UnhydratedScalaDataset, dateRange).withRemoteReadPolicy(ExplicitLocation(ProcAtla))

.toTypedPipe

}

def flatTweetsSource(

implicit dateRange: DateRange

): TypedPipe[UnhydratedFlatTweet] = {

DAL

.read(UnhydratedFlatScalaDataset, dateRange)

.withRemoteReadPolicy(ExplicitLocation(ProcAtla))

.toTypedPipe

}

def userTweetFavoritesSource(

implicit dateRange: DateRange

): TypedPipe[(UserId, TweetId, Timestamp)] = {

DAL

.read(TimelineServiceFavoritesScalaDataset, dateRange)

.withRemoteReadPolicy(ExplicitLocation(ProcAtla))

.toTypedPipe

.flatMap { cfe: ContextualizedFavoriteEvent =>

cfe.event match {

case FavoriteEventUnion.Favorite(fav) =>

Some(fav.userId, fav.tweetId, fav.eventTimeMs)

case \_ =>

None

}

}

}

def userTweetImpressionsSource(

dwellSec: Int = 1

)(

implicit dateRange: DateRange

): TypedPipe[(UserId, TweetId, Timestamp)] = {

DAL

.read(UserInteractionScalaDataset, dateRange)

.withRemoteReadPolicy(AllowCrossClusterSameDC)

.toTypedPipe

.flatMap {

case userInteraction

if userInteraction.interactionType == InteractionType.TweetImpressions =>

userInteraction.interactionDetails match {

case InteractionDetails.TweetImpressionDetails(

TweetImpressionDetails(tweetId, \_, dwellTimeInSecOpt))

if dwellTimeInSecOpt.exists(\_ >= dwellSec) =>

Some(userInteraction.userId, tweetId, userInteraction.timeStamp)

case \_ =>

None

}

case \_ => None

}

}

def transformFavEdges(

input: TypedPipe[EdgeWithDecayedWeights],

halfLifeInDaysForFavScore: Int

)(

implicit uniqueID: UniqueID

): TypedPipe[(Long, Long, Double)] = {

val numEdgesWithSpecifiedHalfLife = Stat(

s"num\_edges\_with\_specified\_half\_life\_${halfLifeInDaysForFavScore}\_days")

val numEdgesWithoutSpecifiedHalfLife = Stat(

s"num\_edges\_without\_specified\_half\_life\_${halfLifeInDaysForFavScore}\_days")

input

.flatMap { edge =>

if (edge.weights.halfLifeInDaysToDecayedSums.contains(halfLifeInDaysForFavScore)) {

numEdgesWithSpecifiedHalfLife.inc()

Some((edge.sourceId, edge.destinationId, edge.weights.halfLifeInDaysToDecayedSums(100)))

} else {

numEdgesWithoutSpecifiedHalfLife.inc()

None

}

}

}

def getFavEdges(

halfLifeInDaysForFavScore: Int

)(

implicit dateRange: DateRange,

uniqueID: UniqueID

): TypedPipe[(Long, Long, Double)] = {

implicit val tz: java.util.TimeZone = DateOps.UTC

transformFavEdges(

DAL

.readMostRecentSnapshotNoOlderThan(UserUserFavGraphScalaDataset, Days(14))

.withRemoteReadPolicy(ExplicitLocation(ProcAtla))

.toTypedPipe,

halfLifeInDaysForFavScore

)

}

def flockReportAsSpamSource(

)(

implicit dateRange: DateRange

): TypedPipe[(UserId, UserId)] = {

DAL

.readMostRecentSnapshot(FlockReportAsSpamEdgesScalaDataset)

.toTypedPipe

.collect {

case edge if edge.state == ValidFlockEdgeStateId =>

(edge.sourceId, edge.destinationId)

}

}

def flockBlocksSource(

)(

implicit dateRange: DateRange

): TypedPipe[(UserId, UserId)] = {

DAL

.readMostRecentSnapshot(FlockBlocksEdgesScalaDataset)

.toTypedPipe

.collect {

case edge if edge.state == ValidFlockEdgeStateId =>

(edge.sourceId, edge.destinationId)

}

}

def flockFollowsSource(

)(

implicit dateRange: DateRange

): TypedPipe[(UserId, UserId)] = {

DAL

.readMostRecentSnapshot(FlockFollowsEdgesScalaDataset)

.toTypedPipe

.collect {

case edge if edge.state == ValidFlockEdgeStateId =>

(edge.sourceId, edge.destinationId)

}

}

def flockReportAsAbuseSource(

)(

implicit dateRange: DateRange

): TypedPipe[(UserId, UserId)] = {

DAL

.readMostRecentSnapshot(FlockReportAsAbuseEdgesScalaDataset)

.toTypedPipe

.collect {

case edge if edge.state == ValidFlockEdgeStateId =>

(edge.sourceId, edge.destinationId)

}

}

def magicRecsNotficationOpenOrClickEventsSource(

implicit dateRange: DateRange

): TypedPipe[MagicRecsNotificationLite] = {

DAL

.read(MagicrecsNotificationLite1DayLagScalaDataset, dateRange)

.toTypedPipe

.filter { entry =>

// keep entries with a valid userId and tweetId, opened or clicked timestamp defined

val userIdExists = entry.targetUserId.isDefined

val tweetIdExists = entry.tweetId.isDefined

val openOrClickExists =

entry.openTimestampMs.isDefined || entry.ntabClickTimestampMs.isDefined

userIdExists && tweetIdExists && openOrClickExists

}

}

def ieSourceTweetEngagementsSource(implicit dateRange: DateRange): TypedPipe[InteractionEvent] = {

DAL

.read(

com.twitter.iesource.processing.events.batch.ServerEngagementsScalaDataset,

dateRange).withColumns(

Set("targetId", "targetType", "engagingUserId", "details", "referenceTweet"))

.toTypedPipe

.filter { event =>

// filter out logged out users because their favorites are less reliable

event.engagingUserId > 0L && event.targetType == InteractionTargetType.Tweet

}

}

private def userIdFromBlenderAdaptiveScribeLog(

blenderAdaptiveLog: AdaptiveSearchScribeLog

): Option[Long] = {

blenderAdaptiveLog.versionedCommonHeader match {

case VersionedCommonHeader.CommonHeader(CommonHeader.ServerHeader(serverHeader)) =>

serverHeader.requestInfo match {

case Some(requestInfo) => requestInfo.ids.get(IdType.UserId).map(\_.toLong)

case \_ => None

}

case \_ => None

}

}

def adaptiveSearchScribeLogsSource(

implicit dateRange: DateRange

): TypedPipe[(UserId, String)] = {

val searchData: TypedPipe[AdaptiveSearchScribeLog] =

DAL

.read(AdaptiveSearchScalaDataset, dateRange).toTypedPipe

searchData

.flatMap({ scribeLog: AdaptiveSearchScribeLog =>

for {

userId <- userIdFromBlenderAdaptiveScribeLog(scribeLog)

// filter out logged out search queries

if userId != 0

queryString <- scribeLog.requestLog.flatMap(\_.request).flatMap(\_.rawQuery)

} yield {

(userId, Set(queryString))

}

})

// if a user searches for the same query multiple times, there could be duplicates.

// De-dup them to get the distinct queries searched by a user

.sumByKey

.flatMap {

case (userId, distinctQuerySet) =>

distinctQuerySet.map { query =>

(userId, query)

}

}

}

}