package com.twitter.simclusters\_v2.scalding.evaluation

import com.twitter.core\_workflows.user\_model.thriftscala.CondensedUserState

import com.twitter.core\_workflows.user\_model.thriftscala.UserState

import com.twitter.pluck.source.core\_workflows.user\_model.CondensedUserStateScalaDataset

import com.twitter.scalding.\_

import com.twitter.scalding.source.TypedText

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

import com.twitter.scalding\_internal.job.TwitterExecutionApp

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

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

import scala.util.Random

/\*\*

\* Helper functions to provide user samples by sampling across user states.

\*/

object UserStateUserSampler {

def getSampleUsersByUserState(

userStateSource: TypedPipe[CondensedUserState],

validStates: Seq[UserState],

samplePercentage: Double

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

assert(samplePercentage >= 0 && samplePercentage <= 1)

val validStateSet = validStates.toSet

userStateSource

.collect {

case data if data.userState.isDefined && validStateSet.contains(data.userState.get) =>

(data.userState.get, data.uid)

}

.filter(\_ => Random.nextDouble() <= samplePercentage)

.forceToDisk

}

/\*\*

\* Given a list of string corresponding to user states, convert them to the UserState type.

\* If the input is empty, default to return all available user states

\*/

def parseUserStates(strStates: Seq[String]): Seq[UserState] = {

if (strStates.isEmpty) {

UserState.list

} else {

strStates.map { str =>

UserState

.valueOf(str).getOrElse(

throw new IllegalArgumentException(

s"Input user\_states $str is invalid. Valid states are: " + UserState.list

)

)

}

}

}

}

/\*\*

\* A variation of the evaluation base where target users are sampled by user states.

\* For each user state of interest (e.x. HEAVY\_TWEETER), we run a separate evaluation call, and

\* output the evaluation results per user state. This is helpful when we want to horizontally

\* compare how users in different user states respond to the candidate tweets.

\*/

trait UserStateBasedEvaluationExecutionBase

extends CandidateEvaluationBase

with TwitterExecutionApp {

def referenceTweets: TypedPipe[ReferenceTweets]

def candidateTweets: TypedPipe[CandidateTweets]

override def job: Execution[Unit] = {

Execution.withId { implicit uniqueId =>

Execution.withArgs { args =>

implicit val dateRange: DateRange =

DateRange.parse(args.list("date"))(DateOps.UTC, DateParser.default)

val outputRootDir = args("outputDir")

val userStates: Seq[UserState] =

UserStateUserSampler.parseUserStates(args.list("user\_states"))

val sampleRate = args.double("sample\_rate")

// For each user state we are interested in, run separate executions and write

// the output into individual sub directories

val userStateSource = DAL.read(CondensedUserStateScalaDataset).toTypedPipe

val userIdsByState =

UserStateUserSampler.getSampleUsersByUserState(userStateSource, userStates, sampleRate)

val executionsPerUserState = userStates.map { userState =>

val sampleUsers = userIdsByState.collect { case data if data.\_1 == userState => data.\_2 }

val outputPath = outputRootDir + "/" + userState + "/"

super

.runSampledEvaluation(sampleUsers, referenceTweets, candidateTweets)

.writeExecution(TypedText.csv(outputPath))

}

// Run evaluation for each user state in parallel

Execution.sequence(executionsPerUserState).unit

}

}

}

}

/\*\*

\* A basic flow for evaluating the quality of a set of candidate tweets, typically generated by an

\* algorithm (ex. SimClusters), by comparing its engagement rates against a set of reference tweets

\* The job goes through the following steps:

\* 1. Generate a group of target users on which we measure tweet engagements

\* 2. Collect tweets impressed by these users and their engagements on tweets from a labeled

\* tweet source (ex. Home Timeline engagement data), and form a reference set

\* 3. For each candidate tweet, collect the engagement rates from the reference set

\* 4. Run evaluation calculations (ex. percentage of intersection, engagement rate, etc)

\*

\* Each sub class is expected to provide 3 sets of data sources, which are the sample users,

\* candidate tweet sources, and reference tweet sources.

\*/

trait CandidateEvaluationBase {

private def getSampledReferenceTweets(

referenceTweetEngagements: TypedPipe[ReferenceTweets],

sampleUsers: TypedPipe[Long]

): TypedPipe[ReferenceTweets] = {

referenceTweetEngagements

.groupBy(\_.targetUserId)

.join(sampleUsers.asKeys)

.map { case (targetUserId, (referenceEngagements, \_)) => referenceEngagements }

}

private def getSampledCandidateTweets(

candidateTweets: TypedPipe[CandidateTweets],

sampleUsers: TypedPipe[Long]

): TypedPipe[CandidateTweets] = {

candidateTweets

.groupBy(\_.targetUserId)

.join(sampleUsers.asKeys)

.map { case (\_, (tweets, \_)) => tweets }

}

/\*\*

\* Evaluation function, should be overridden by implementing sub classes to suit individual

\* objectives, such as like engagement rates, CRT, etc.

\* @param sampledReference

\* @param sampledCandidate

\*/

def evaluateResults(

sampledReference: TypedPipe[ReferenceTweets],

sampledCandidate: TypedPipe[CandidateTweets]

): TypedPipe[String]

/\*\*

\* Given a list of target users, the reference tweet set, and the candidate tweet set,

\* calculate the engagement rates on the reference set and the candidate set by these users.

\* The evaluation result should be converted into an itemized format

\* these users.

\* @param referenceTweets

\* @param candidateTweets

\* @return

\*/

def runSampledEvaluation(

targetUserSamples: TypedPipe[Long],

referenceTweets: TypedPipe[ReferenceTweets],

candidateTweets: TypedPipe[CandidateTweets]

): TypedPipe[String] = {

val sampledCandidate = getSampledCandidateTweets(candidateTweets, targetUserSamples)

val referencePerUser = getSampledReferenceTweets(referenceTweets, targetUserSamples)

evaluateResults(referencePerUser, sampledCandidate)

}

}