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

import com.twitter.scalding.{Execution, TypedPipe, UniqueID}

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

CandidateTweet,

CandidateTweets,

ReferenceTweet,

ReferenceTweets,

TweetLabels

}

import com.twitter.algebird.Aggregator.size

import com.twitter.scalding.typed.{CoGrouped, ValuePipe}

import com.twitter.util.TwitterDateFormat

import java.util.Calendar

/\*\*

\* Statistics about the number of users who have engaged with tweets

\*/

case class UserEngagerCounts(

numDistinctTargetUsers: Long,

numDistinctLikeEngagers: Long,

numDistinctRetweetEngagers: Long)

/\*\*

\* Tweet side statistics, e.x. number of tweets, authors, etc.

\*/

case class TweetStats(

numTweets: Long,

numDistinctTweets: Long,

numDistinctAuthors: Option[Long],

avgScore: Option[Double])

/\*\*

\* Helper data container class for storing engagement counts

\*/

case class TweetEngagementCounts(like: Long, retweet: Long, click: Long, hasEngagement: Long)

/\*\*

\* Helper data container class for storing engagement rates

\*/

case class TweetEngagementRates(like: Double, retweet: Double, click: Double, hasEngagement: Double)

case class LabelCorrelations(

pearsonCoefficientForLikes: Double,

cosineSimilarityGlobal: Double,

cosineSimilarityPerUserAvg: Double) {

private val f = java.text.NumberFormat.getInstance

def format(): String = {

Seq(

s"\tPearson Coefficient: ${f.format(pearsonCoefficientForLikes)}",

s"\tCosine similarity: ${f.format(cosineSimilarityGlobal)}",

s"\tAverage cosine similarity for all users: ${f.format(cosineSimilarityPerUserAvg)}"

).mkString("\n")

}

}

/\*\*

\* Helper tweet data container that can hold both the reference label engagements as well as the

\* recommendation algorithm's scores. Helpful for evaluating joint data

\*/

case class LabeledTweet(

targetUserId: Long,

tweetId: Long,

authorId: Long,

labels: TweetLabels,

algorithmScore: Option[Double])

case class LabeledTweetsResults(

tweetStats: TweetStats,

userEngagerCounts: UserEngagerCounts,

tweetEngagementCounts: TweetEngagementCounts,

tweetEngagementRates: TweetEngagementRates,

labelCorrelations: Option[LabelCorrelations] = None) {

private val f = java.text.NumberFormat.getInstance

def format(title: String = ""): String = {

val str = Seq(

s"Number of tweets: ${f.format(tweetStats.numTweets)}",

s"Number of distinct tweets: ${f.format(tweetStats.numDistinctTweets)}",

s"Number of distinct users targeted: ${f.format(userEngagerCounts.numDistinctTargetUsers)}",

s"Number of distinct authors: ${tweetStats.numDistinctAuthors.map(f.format).getOrElse("N/A")}",

s"Average algorithm score of tweets: ${tweetStats.avgScore.map(f.format).getOrElse("N/A")}",

s"Engager counts:",

s"\tNumber of users who liked tweets: ${f.format(userEngagerCounts.numDistinctLikeEngagers)}",

s"\tNumber of users who retweeted tweets: ${f.format(userEngagerCounts.numDistinctRetweetEngagers)}",

s"Tweet engagement counts:",

s"\tNumber of Likes: ${f.format(tweetEngagementCounts.like)}",

s"\tNumber of Retweets: ${f.format(tweetEngagementCounts.retweet)}",

s"\tNumber of Clicks: ${f.format(tweetEngagementCounts.click)}",

s"\tNumber of tweets with any engagements: ${f.format(tweetEngagementCounts.hasEngagement)}",

s"Tweet engagement rates:",

s"\tRate of Likes: ${f.format(tweetEngagementRates.like \* 100)}%",

s"\tRate of Retweets: ${f.format(tweetEngagementRates.retweet \* 100)}%",

s"\tRate of Clicks: ${f.format(tweetEngagementRates.click \* 100)}%",

s"\tRate of any engagement: ${f.format(tweetEngagementRates.hasEngagement \* 100)}%"

).mkString("\n")

val correlations = labelCorrelations.map("\n" + \_.format()).getOrElse("")

s"$title\n$str$correlations"

}

}

case class CandidateResults(tweetStats: TweetStats, numDistinctTargetUsers: Long) {

private val f = java.text.NumberFormat.getInstance

def format(title: String = ""): String = {

val str = Seq(

s"Number of tweets: ${f.format(tweetStats.numTweets)}",

s"Number of distinct tweets: ${f.format(tweetStats.numDistinctTweets)}",

s"Number of distinct users targeted: ${f.format(numDistinctTargetUsers)}",

s"Number of distinct authors: ${tweetStats.numDistinctAuthors.map(f.format).getOrElse("N/A")}",

s"Average algorithm score of tweets: ${tweetStats.avgScore.map(f.format).getOrElse("N/A")}"

).mkString("\n")

s"$title\n$str"

}

}

/\*\*

\* Helper class for evaluating a given candidate tweet set against a reference tweet set.

\* It provides aggregation evaluation metrics such as sum of engagements, rate of engagements, etc.

\*/

object EvaluationMetricHelper {

private def toLong(bool: Boolean): Long = {

if (bool) 1L else 0L

}

/\*\*

\* Core engagements are user actions that count towards core metrics, e.x. like, RT, etc

\*/

private def hasCoreEngagements(labels: TweetLabels): Boolean = {

labels.isRetweeted ||

labels.isLiked ||

labels.isQuoted ||

labels.isReplied

}

/\*\*

\* Whether there are core engagements or click on the tweet

\*/

private def hasCoreEngagementsOrClick(labels: TweetLabels): Boolean = {

hasCoreEngagements(labels) || labels.isClicked

}

/\*\*

\* Return outer join of reference tweets and candidate tweets, keyed by (targetUserId, tweetId).

\* The output of this can then be reused to fetch the inner join / left / right join,

\* without having to redo the expensive join

\*

\* NOTE: Assumes the uniqueness of keys (i.e. (targetId, tweetId)). Make sure to dedup tweetIds

\* for each targetId, otherwise .join() will yield duplicate results.

\*/

def outerJoinReferenceAndCandidate(

referencePipe: TypedPipe[ReferenceTweets],

candidatePipe: TypedPipe[CandidateTweets]

): CoGrouped[(Long, Long), (Option[ReferenceTweet], Option[CandidateTweet])] = {

val references = referencePipe

.flatMap { refTweets =>

refTweets.impressedTweets.map { refTweet =>

((refTweets.targetUserId, refTweet.tweetId), refTweet)

}

}

val candidates = candidatePipe

.flatMap { candTweets =>

candTweets.recommendedTweets.map { candTweet =>

((candTweets.targetUserId, candTweet.tweetId), candTweet)

}

}

references.outerJoin(candidates).withReducers(50)

}

/\*\*

\* Convert reference tweets to labeled tweets. We do this so that we can re-use the common

\* metric calculations for labeled tweets on reference tweets

\*/

def getLabeledReference(referencePipe: TypedPipe[ReferenceTweets]): TypedPipe[LabeledTweet] = {

referencePipe

.flatMap { refTweets =>

refTweets.impressedTweets.map { tweet =>

// Reference tweets do not have scores

LabeledTweet(refTweets.targetUserId, tweet.tweetId, tweet.authorId, tweet.labels, None)

}

}

}

def getUniqueCount[T](pipe: TypedPipe[T])(implicit ord: scala.Ordering[T]): Execution[Long] = {

pipe.distinct

.aggregate(size)

.toOptionExecution

.map(\_.getOrElse(0L))

}

def countUniqueEngagedUsersBy(

labeledTweetsPipe: TypedPipe[LabeledTweet],

f: TweetLabels => Boolean

): Execution[Long] = {

getUniqueCount[Long](labeledTweetsPipe.collect { case t if f(t.labels) => t.targetUserId })

}

def countUniqueLabeledTargetUsers(labeledTweetsPipe: TypedPipe[LabeledTweet]): Execution[Long] = {

getUniqueCount[Long](labeledTweetsPipe.map(\_.targetUserId))

}

def countUniqueCandTargetUsers(candidatePipe: TypedPipe[CandidateTweets]): Execution[Long] = {

getUniqueCount[Long](candidatePipe.map(\_.targetUserId))

}

def countUniqueLabeledAuthors(labeledTweetPipe: TypedPipe[LabeledTweet]): Execution[Long] = {

getUniqueCount[Long](labeledTweetPipe.map(\_.authorId))

}

/\*\*

\* Helper function to calculate the basic engagement rates

\*/

def getEngagementRate(

basicStats: TweetStats,

engagementCount: TweetEngagementCounts

): TweetEngagementRates = {

val numTweets = basicStats.numTweets.toDouble

if (numTweets <= 0) throw new IllegalArgumentException("Invalid tweet counts")

val likeRate = engagementCount.like / numTweets

val rtRate = engagementCount.retweet / numTweets

val clickRate = engagementCount.click / numTweets

val engagementRate = engagementCount.hasEngagement / numTweets

TweetEngagementRates(likeRate, rtRate, clickRate, engagementRate)

}

/\*\*

\* Helper function to calculate the basic stats for a pipe of candidate tweets

\*/

def getTweetStatsForCandidateExec(

candidatePipe: TypedPipe[CandidateTweets]

): Execution[TweetStats] = {

val pipe = candidatePipe.map { candTweets =>

(candTweets.targetUserId, candTweets.recommendedTweets)

}.sumByKey // Dedup by targetId, in case there exists multiple entries.

val distinctTweetPipe = pipe.flatMap(\_.\_2.map(\_.tweetId)).distinct.aggregate(size)

val otherStats = pipe

.map {

case (uid, recommendedTweets) =>

val scoreSum = recommendedTweets.flatMap(\_.score).sum

(recommendedTweets.size.toLong, scoreSum)

}

.sum

.map {

case (numTweets, scoreSum) =>

if (numTweets <= 0) throw new IllegalArgumentException("Invalid tweet counts")

val avgScore = scoreSum / numTweets.toDouble

(numTweets, avgScore)

}

ValuePipe

.fold(distinctTweetPipe, otherStats) {

case (numDistinctTweet, (numTweets, avgScore)) =>

// no author side information for candidate tweets yet

TweetStats(numTweets, numDistinctTweet, None, Some(avgScore))

}.getOrElseExecution(TweetStats(0L, 0L, None, None))

}

/\*\*

\* Helper function to count the total number of engagements

\*/

def getLabeledEngagementCountExec(

labeledTweets: TypedPipe[LabeledTweet]

): Execution[TweetEngagementCounts] = {

labeledTweets

.map { labeledTweet =>

val like = toLong(labeledTweet.labels.isLiked)

val retweet = toLong(labeledTweet.labels.isRetweeted)

val click = toLong(labeledTweet.labels.isClicked)

val hasEngagement = toLong(hasCoreEngagementsOrClick(labeledTweet.labels))

(like, retweet, click, hasEngagement)

}

.sum

.map {

case (like, retweet, click, hasEngagement) =>

TweetEngagementCounts(like, retweet, click, hasEngagement)

}

.getOrElseExecution(TweetEngagementCounts(0L, 0L, 0L, 0L))

}

/\*\*

\* Count the total number of unique users who have engaged with tweets

\*/

def getTargetUserStatsForLabeledTweetsExec(

labeledTweetsPipe: TypedPipe[LabeledTweet]

): Execution[UserEngagerCounts] = {

val numUniqueTargetUsersExec = countUniqueLabeledTargetUsers(labeledTweetsPipe)

val numUniqueLikeUsersExec =

countUniqueEngagedUsersBy(labeledTweetsPipe, labels => labels.isLiked)

val numUniqueRetweetUsersExec =

countUniqueEngagedUsersBy(labeledTweetsPipe, labels => labels.isRetweeted)

Execution

.zip(

numUniqueTargetUsersExec,

numUniqueLikeUsersExec,

numUniqueRetweetUsersExec

)

.map {

case (numTarget, like, retweet) =>

UserEngagerCounts(

numDistinctTargetUsers = numTarget,

numDistinctLikeEngagers = like,

numDistinctRetweetEngagers = retweet

)

}

}

/\*\*

\* Helper function to calculate the basic stats for a pipe of labeled tweets.

\*/

def getTweetStatsForLabeledTweetsExec(

labeledTweetPipe: TypedPipe[LabeledTweet]

): Execution[TweetStats] = {

val uniqueAuthorsExec = countUniqueLabeledAuthors(labeledTweetPipe)

val uniqueTweetExec =

labeledTweetPipe.map(\_.tweetId).distinct.aggregate(size).getOrElseExecution(0L)

val scoresExec = labeledTweetPipe

.map { t => (t.targetUserId, (1, t.algorithmScore.getOrElse(0.0))) }

.sumByKey // Dedup by targetId, in case there exists multiple entries.

.map {

case (uid, (c1, c2)) =>

(c1.toLong, c2)

}

.sum

.map {

case (numTweets, scoreSum) =>

if (numTweets <= 0) throw new IllegalArgumentException("Invalid tweet counts")

val avgScore = scoreSum / numTweets.toDouble

(numTweets, Option(avgScore))

}

.getOrElseExecution((0L, None))

Execution

.zip(uniqueAuthorsExec, uniqueTweetExec, scoresExec)

.map {

case (numDistinctAuthors, numUniqueTweets, (numTweets, avgScores)) =>

TweetStats(numTweets, numUniqueTweets, Some(numDistinctAuthors), avgScores)

}

}

/\*\*

\* Print a update message to the stdout when a step is done.

\*/

private def printOnCompleteMsg(stepDescription: String, startTimeMillis: Long): Unit = {

val formatDate = TwitterDateFormat("yyyy-MM-dd hh:mm:ss")

val now = Calendar.getInstance().getTime

val secondsSpent = (now.getTime - startTimeMillis) / 1000

println(

s"- ${formatDate.format(now)}\tStep complete: $stepDescription\t " +

s"Time spent: ${secondsSpent / 60}m${secondsSpent % 60}s"

)

}

/\*\*

\* Calculate the metrics of a pipe of [[CandidateTweets]]

\*/

private def getEvaluationResultsForCandidates(

candidatePipe: TypedPipe[CandidateTweets]

): Execution[CandidateResults] = {

val tweetStatsExec = getTweetStatsForCandidateExec(candidatePipe)

val numDistinctTargetUsersExec = countUniqueCandTargetUsers(candidatePipe)

Execution

.zip(tweetStatsExec, numDistinctTargetUsersExec)

.map {

case (tweetStats, numDistinctTargetUsers) =>

CandidateResults(tweetStats, numDistinctTargetUsers)

}

}

/\*\*

\* Calculate the metrics of a pipe of [[LabeledTweet]]

\*/

private def getEvaluationResultsForLabeledTweets(

labeledTweetPipe: TypedPipe[LabeledTweet],

getLabelCorrelations: Boolean = false

): Execution[LabeledTweetsResults] = {

val tweetStatsExec = getTweetStatsForLabeledTweetsExec(labeledTweetPipe)

val userStatsExec = getTargetUserStatsForLabeledTweetsExec(labeledTweetPipe)

val engagementCountExec = getLabeledEngagementCountExec(labeledTweetPipe)

val correlationsExec = if (getLabelCorrelations) {

Execution

.zip(

LabelCorrelationsHelper.pearsonCoefficientForLike(labeledTweetPipe),

LabelCorrelationsHelper.cosineSimilarityForLike(labeledTweetPipe),

LabelCorrelationsHelper.cosineSimilarityForLikePerUser(labeledTweetPipe)

).map {

case (pearsonCoeff, globalCos, avgCos) =>

Some(LabelCorrelations(pearsonCoeff, globalCos, avgCos))

}

} else {

ValuePipe(None).getOrElseExecution(None) // Empty pipe with a None value

}

Execution

.zip(tweetStatsExec, engagementCountExec, userStatsExec, correlationsExec)

.map {

case (tweetStats, engagementCount, engagerCount, correlationsOpt) =>

val engagementRate = getEngagementRate(tweetStats, engagementCount)

LabeledTweetsResults(

tweetStats,

engagerCount,

engagementCount,

engagementRate,

correlationsOpt)

}

}

private def runAllEvalForCandidates(

candidatePipe: TypedPipe[CandidateTweets],

outerJoinPipe: TypedPipe[((Long, Long), (Option[ReferenceTweet], Option[CandidateTweet]))]

): Execution[(CandidateResults, CandidateResults)] = {

val t0 = System.currentTimeMillis()

val candidateNotInIntersectionPipe =

outerJoinPipe

.collect {

case ((targetUserId, \_), (None, Some(candTweet))) => (targetUserId, Seq(candTweet))

}

.sumByKey

.map { case (targetUserId, candTweets) => CandidateTweets(targetUserId, candTweets) }

.forceToDisk

Execution

.zip(

getEvaluationResultsForCandidates(candidatePipe),

getEvaluationResultsForCandidates(candidateNotInIntersectionPipe)

).onComplete(\_ => printOnCompleteMsg("runAllEvalForCandidates()", t0))

}

private def runAllEvalForIntersection(

outerJoinPipe: TypedPipe[((Long, Long), (Option[ReferenceTweet], Option[CandidateTweet]))]

)(

implicit uniqueID: UniqueID

): Execution[(LabeledTweetsResults, LabeledTweetsResults, LabeledTweetsResults)] = {

val t0 = System.currentTimeMillis()

val intersectionTweetsPipe = outerJoinPipe.collect {

case ((targetUserId, tweetId), (Some(refTweet), Some(candTweet))) =>

LabeledTweet(targetUserId, tweetId, refTweet.authorId, refTweet.labels, candTweet.score)

}.forceToDisk

val likedTweetsPipe = intersectionTweetsPipe.filter(\_.labels.isLiked)

val notLikedTweetsPipe = intersectionTweetsPipe.filter(!\_.labels.isLiked)

Execution

.zip(

getEvaluationResultsForLabeledTweets(intersectionTweetsPipe, getLabelCorrelations = true),

getEvaluationResultsForLabeledTweets(likedTweetsPipe),

getEvaluationResultsForLabeledTweets(notLikedTweetsPipe)

).onComplete(\_ => printOnCompleteMsg("runAllEvalForIntersection()", t0))

}

private def runAllEvalForReferences(

referencePipe: TypedPipe[ReferenceTweets],

outerJoinPipe: TypedPipe[((Long, Long), (Option[ReferenceTweet], Option[CandidateTweet]))]

): Execution[(LabeledTweetsResults, LabeledTweetsResults)] = {

val t0 = System.currentTimeMillis()

val labeledReferenceNotInIntersectionPipe =

outerJoinPipe.collect {

case ((targetUserId, \_), (Some(refTweet), None)) =>

LabeledTweet(targetUserId, refTweet.tweetId, refTweet.authorId, refTweet.labels, None)

}.forceToDisk

Execution

.zip(

getEvaluationResultsForLabeledTweets(getLabeledReference(referencePipe)),

getEvaluationResultsForLabeledTweets(labeledReferenceNotInIntersectionPipe)

).onComplete(\_ => printOnCompleteMsg("runAllEvalForReferences()", t0))

}

def runAllEvaluations(

referencePipe: TypedPipe[ReferenceTweets],

candidatePipe: TypedPipe[CandidateTweets]

)(

implicit uniqueID: UniqueID

): Execution[String] = {

val t0 = System.currentTimeMillis()

// Force everything to disk to maximize data re-use

Execution

.zip(

referencePipe.forceToDiskExecution,

candidatePipe.forceToDiskExecution

).flatMap {

case (referenceDiskPipe, candidateDiskPipe) =>

outerJoinReferenceAndCandidate(referenceDiskPipe, candidateDiskPipe).forceToDiskExecution

.flatMap { outerJoinPipe =>

val referenceResultsExec = runAllEvalForReferences(referenceDiskPipe, outerJoinPipe)

val intersectionResultsExec = runAllEvalForIntersection(outerJoinPipe)

val candidateResultsExec = runAllEvalForCandidates(candidateDiskPipe, outerJoinPipe)

Execution

.zip(

referenceResultsExec,

intersectionResultsExec,

candidateResultsExec

).map {

case (

(allReference, referenceNotInIntersection),

(allIntersection, intersectionLiked, intersectionNotLiked),

(allCandidate, candidateNotInIntersection)) =>

val timeSpent = (System.currentTimeMillis() - t0) / 1000

val resultStr = Seq(

"===================================================",

s"Evaluation complete. Took ${timeSpent / 60}m${timeSpent % 60}s ",

allReference.format("-----Metrics for all Reference Tweets-----"),

referenceNotInIntersection.format(

"-----Metrics for Reference Tweets that are not in the intersection-----"

),

allIntersection.format("-----Metrics for all Intersection Tweets-----"),

intersectionLiked.format("-----Metrics for Liked Intersection Tweets-----"),

intersectionNotLiked.format(

"-----Metrics for not Liked Intersection Tweets-----"),

allCandidate.format("-----Metrics for all Candidate Tweets-----"),

candidateNotInIntersection.format(

"-----Metrics for Candidate Tweets that are not in the intersection-----"

),

"===================================================\n"

).mkString("\n")

println(resultStr)

resultStr

}

.onComplete(\_ =>

printOnCompleteMsg(

"Evaluation complete. Check stdout or output logs for results.",

t0))

}

}

}

}