package com.twitter.representationscorer.twistlyfeatures

import com.twitter.finagle.stats.Counter

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.representationscorer.common.TweetId

import com.twitter.representationscorer.common.UserId

import com.twitter.representationscorer.scorestore.ScoreStore

import com.twitter.representationscorer.thriftscala.SimClustersRecentEngagementSimilarities

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

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

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

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

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

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

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

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

import com.twitter.stitch.Stitch

import javax.inject.Inject

class Scorer @Inject() (

fetchEngagementsFromUSS: Long => Stitch[Engagements],

scoreStore: ScoreStore,

stats: StatsReceiver) {

import Scorer.\_

private val scoreStats = stats.scope("score")

private val scoreCalculationStats = scoreStats.scope("calculation")

private val scoreResultStats = scoreStats.scope("result")

private val scoresNonEmptyCounter = scoreResultStats.scope("all").counter("nonEmpty")

private val scoresNonZeroCounter = scoreResultStats.scope("all").counter("nonZero")

private val tweetScoreStats = scoreCalculationStats.scope("tweetScore").stat("latency")

private val userScoreStats = scoreCalculationStats.scope("userScore").stat("latency")

private val favNonZero = scoreResultStats.scope("favs").counter("nonZero")

private val favNonEmpty = scoreResultStats.scope("favs").counter("nonEmpty")

private val retweetsNonZero = scoreResultStats.scope("retweets").counter("nonZero")

private val retweetsNonEmpty = scoreResultStats.scope("retweets").counter("nonEmpty")

private val followsNonZero = scoreResultStats.scope("follows").counter("nonZero")

private val followsNonEmpty = scoreResultStats.scope("follows").counter("nonEmpty")

private val sharesNonZero = scoreResultStats.scope("shares").counter("nonZero")

private val sharesNonEmpty = scoreResultStats.scope("shares").counter("nonEmpty")

private val repliesNonZero = scoreResultStats.scope("replies").counter("nonZero")

private val repliesNonEmpty = scoreResultStats.scope("replies").counter("nonEmpty")

private val originalTweetsNonZero = scoreResultStats.scope("originalTweets").counter("nonZero")

private val originalTweetsNonEmpty = scoreResultStats.scope("originalTweets").counter("nonEmpty")

private val videoViewsNonZero = scoreResultStats.scope("videoViews").counter("nonZero")

private val videoViewsNonEmpty = scoreResultStats.scope("videoViews").counter("nonEmpty")

private val blockNonZero = scoreResultStats.scope("block").counter("nonZero")

private val blockNonEmpty = scoreResultStats.scope("block").counter("nonEmpty")

private val muteNonZero = scoreResultStats.scope("mute").counter("nonZero")

private val muteNonEmpty = scoreResultStats.scope("mute").counter("nonEmpty")

private val reportNonZero = scoreResultStats.scope("report").counter("nonZero")

private val reportNonEmpty = scoreResultStats.scope("report").counter("nonEmpty")

private val dontlikeNonZero = scoreResultStats.scope("dontlike").counter("nonZero")

private val dontlikeNonEmpty = scoreResultStats.scope("dontlike").counter("nonEmpty")

private val seeFewerNonZero = scoreResultStats.scope("seeFewer").counter("nonZero")

private val seeFewerNonEmpty = scoreResultStats.scope("seeFewer").counter("nonEmpty")

private def getTweetScores(

candidateTweetId: TweetId,

sourceTweetIds: Seq[TweetId]

): Stitch[Seq[ScoreResult]] = {

val getScoresStitch = Stitch.traverse(sourceTweetIds) { sourceTweetId =>

scoreStore

.uniformScoringStoreStitch(getTweetScoreId(sourceTweetId, candidateTweetId))

.liftNotFoundToOption

.map(score => ScoreResult(sourceTweetId, score.map(\_.score)))

}

Stitch.time(getScoresStitch).flatMap {

case (tryResult, duration) =>

tweetScoreStats.add(duration.inMillis)

Stitch.const(tryResult)

}

}

private def getUserScores(

tweetId: TweetId,

authorIds: Seq[UserId]

): Stitch[Seq[ScoreResult]] = {

val getScoresStitch = Stitch.traverse(authorIds) { authorId =>

scoreStore

.uniformScoringStoreStitch(getAuthorScoreId(authorId, tweetId))

.liftNotFoundToOption

.map(score => ScoreResult(authorId, score.map(\_.score)))

}

Stitch.time(getScoresStitch).flatMap {

case (tryResult, duration) =>

userScoreStats.add(duration.inMillis)

Stitch.const(tryResult)

}

}

/\*\*

\* Get the [[SimClustersRecentEngagementSimilarities]] result containing the similarity

\* features for the given userId-TweetId.

\*/

def get(

userId: UserId,

tweetId: TweetId

): Stitch[SimClustersRecentEngagementSimilarities] = {

get(userId, Seq(tweetId)).map(x => x.head)

}

/\*\*

\* Get a list of [[SimClustersRecentEngagementSimilarities]] results containing the similarity

\* features for the given tweets of the user Id.

\* Guaranteed to be the same number/order as requested.

\*/

def get(

userId: UserId,

tweetIds: Seq[TweetId]

): Stitch[Seq[SimClustersRecentEngagementSimilarities]] = {

fetchEngagementsFromUSS(userId)

.flatMap(engagements => {

// For each tweet received in the request, compute the similarity scores between them

// and the user signals fetched from USS.

Stitch

.join(

Stitch.traverse(tweetIds)(id => getTweetScores(id, engagements.tweetIds)),

Stitch.traverse(tweetIds)(id => getUserScores(id, engagements.authorIds)),

)

.map {

case (tweetScoresSeq, userScoreSeq) =>

// All seq have = size because when scores don't exist, they are returned as Option

(tweetScoresSeq, userScoreSeq).zipped.map { (tweetScores, userScores) =>

computeSimilarityScoresPerTweet(

engagements,

tweetScores.groupBy(\_.id),

userScores.groupBy(\_.id))

}

}

})

}

/\*\*

\*

\* Computes the [[SimClustersRecentEngagementSimilarities]]

\* using the given tweet-tweet and user-tweet scores in TweetScoresMap

\* and the user signals in [[Engagements]].

\*/

private def computeSimilarityScoresPerTweet(

engagements: Engagements,

tweetScores: Map[TweetId, Seq[ScoreResult]],

authorScores: Map[UserId, Seq[ScoreResult]]

): SimClustersRecentEngagementSimilarities = {

val favs7d = engagements.favs7d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val favs1d = engagements.favs1d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val retweets7d = engagements.retweets7d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val retweets1d = engagements.retweets1d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val follows30d = engagements.follows30d.view

.flatMap(s => authorScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val follows7d = engagements.follows7d.view

.flatMap(s => authorScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val shares7d = engagements.shares7d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val shares1d = engagements.shares1d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val replies7d = engagements.replies7d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val replies1d = engagements.replies1d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val originalTweets7d = engagements.originalTweets7d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val originalTweets1d = engagements.originalTweets1d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val videoViews7d = engagements.videoPlaybacks7d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val videoViews1d = engagements.videoPlaybacks1d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val block30d = engagements.block30d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val block7d = engagements.block7d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val block1d = engagements.block1d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val mute30d = engagements.mute30d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val mute7d = engagements.mute7d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val mute1d = engagements.mute1d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val report30d = engagements.report30d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val report7d = engagements.report7d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val report1d = engagements.report1d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val dontlike30d = engagements.dontlike30d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val dontlike7d = engagements.dontlike7d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val dontlike1d = engagements.dontlike1d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val seeFewer30d = engagements.seeFewer30d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val seeFewer7d = engagements.seeFewer7d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val seeFewer1d = engagements.seeFewer1d.view

.flatMap(s => tweetScores.get(s.targetId))

.flatten.flatMap(\_.score)

.force

val result = SimClustersRecentEngagementSimilarities(

fav1dLast10Max = max(favs1d),

fav1dLast10Avg = avg(favs1d),

fav7dLast10Max = max(favs7d),

fav7dLast10Avg = avg(favs7d),

retweet1dLast10Max = max(retweets1d),

retweet1dLast10Avg = avg(retweets1d),

retweet7dLast10Max = max(retweets7d),

retweet7dLast10Avg = avg(retweets7d),

follow7dLast10Max = max(follows7d),

follow7dLast10Avg = avg(follows7d),

follow30dLast10Max = max(follows30d),

follow30dLast10Avg = avg(follows30d),

share1dLast10Max = max(shares1d),

share1dLast10Avg = avg(shares1d),

share7dLast10Max = max(shares7d),

share7dLast10Avg = avg(shares7d),

reply1dLast10Max = max(replies1d),

reply1dLast10Avg = avg(replies1d),

reply7dLast10Max = max(replies7d),

reply7dLast10Avg = avg(replies7d),

originalTweet1dLast10Max = max(originalTweets1d),

originalTweet1dLast10Avg = avg(originalTweets1d),

originalTweet7dLast10Max = max(originalTweets7d),

originalTweet7dLast10Avg = avg(originalTweets7d),

videoPlayback1dLast10Max = max(videoViews1d),

videoPlayback1dLast10Avg = avg(videoViews1d),

videoPlayback7dLast10Max = max(videoViews7d),

videoPlayback7dLast10Avg = avg(videoViews7d),

block1dLast10Max = max(block1d),

block1dLast10Avg = avg(block1d),

block7dLast10Max = max(block7d),

block7dLast10Avg = avg(block7d),

block30dLast10Max = max(block30d),

block30dLast10Avg = avg(block30d),

mute1dLast10Max = max(mute1d),

mute1dLast10Avg = avg(mute1d),

mute7dLast10Max = max(mute7d),

mute7dLast10Avg = avg(mute7d),

mute30dLast10Max = max(mute30d),

mute30dLast10Avg = avg(mute30d),

report1dLast10Max = max(report1d),

report1dLast10Avg = avg(report1d),

report7dLast10Max = max(report7d),

report7dLast10Avg = avg(report7d),

report30dLast10Max = max(report30d),

report30dLast10Avg = avg(report30d),

dontlike1dLast10Max = max(dontlike1d),

dontlike1dLast10Avg = avg(dontlike1d),

dontlike7dLast10Max = max(dontlike7d),

dontlike7dLast10Avg = avg(dontlike7d),

dontlike30dLast10Max = max(dontlike30d),

dontlike30dLast10Avg = avg(dontlike30d),

seeFewer1dLast10Max = max(seeFewer1d),

seeFewer1dLast10Avg = avg(seeFewer1d),

seeFewer7dLast10Max = max(seeFewer7d),

seeFewer7dLast10Avg = avg(seeFewer7d),

seeFewer30dLast10Max = max(seeFewer30d),

seeFewer30dLast10Avg = avg(seeFewer30d),

)

trackStats(result)

result

}

private def trackStats(result: SimClustersRecentEngagementSimilarities): Unit = {

val scores = Seq(

result.fav7dLast10Max,

result.retweet7dLast10Max,

result.follow30dLast10Max,

result.share1dLast10Max,

result.share7dLast10Max,

result.reply7dLast10Max,

result.originalTweet7dLast10Max,

result.videoPlayback7dLast10Max,

result.block30dLast10Max,

result.mute30dLast10Max,

result.report30dLast10Max,

result.dontlike30dLast10Max,

result.seeFewer30dLast10Max

)

val nonEmpty = scores.exists(\_.isDefined)

val nonZero = scores.exists { case Some(score) if score > 0 => true; case \_ => false }

if (nonEmpty) {

scoresNonEmptyCounter.incr()

}

if (nonZero) {

scoresNonZeroCounter.incr()

}

// We use the largest window of a given type of score,

// because the largest window is inclusive of smaller windows.

trackSignalStats(favNonEmpty, favNonZero, result.fav7dLast10Avg)

trackSignalStats(retweetsNonEmpty, retweetsNonZero, result.retweet7dLast10Avg)

trackSignalStats(followsNonEmpty, followsNonZero, result.follow30dLast10Avg)

trackSignalStats(sharesNonEmpty, sharesNonZero, result.share7dLast10Avg)

trackSignalStats(repliesNonEmpty, repliesNonZero, result.reply7dLast10Avg)

trackSignalStats(originalTweetsNonEmpty, originalTweetsNonZero, result.originalTweet7dLast10Avg)

trackSignalStats(videoViewsNonEmpty, videoViewsNonZero, result.videoPlayback7dLast10Avg)

trackSignalStats(blockNonEmpty, blockNonZero, result.block30dLast10Avg)

trackSignalStats(muteNonEmpty, muteNonZero, result.mute30dLast10Avg)

trackSignalStats(reportNonEmpty, reportNonZero, result.report30dLast10Avg)

trackSignalStats(dontlikeNonEmpty, dontlikeNonZero, result.dontlike30dLast10Avg)

trackSignalStats(seeFewerNonEmpty, seeFewerNonZero, result.seeFewer30dLast10Avg)

}

private def trackSignalStats(nonEmpty: Counter, nonZero: Counter, score: Option[Double]): Unit = {

if (score.nonEmpty) {

nonEmpty.incr()

if (score.get > 0)

nonZero.incr()

}

}

}

object Scorer {

def avg(s: Traversable[Double]): Option[Double] =

if (s.isEmpty) None else Some(s.sum / s.size)

def max(s: Traversable[Double]): Option[Double] =

if (s.isEmpty) None else Some(s.foldLeft(0.0D) { (curr, \_max) => math.max(curr, \_max) })

private def getAuthorScoreId(

userId: UserId,

tweetId: TweetId

) = {

ScoreId(

algorithm = ScoringAlgorithm.PairEmbeddingCosineSimilarity,

internalId = ScoreInternalId.SimClustersEmbeddingPairScoreId(

SimClustersEmbeddingPairScoreId(

SimClustersEmbeddingId(

internalId = InternalId.UserId(userId),

modelVersion = ModelVersion.Model20m145k2020,

embeddingType = EmbeddingType.FavBasedProducer

),

SimClustersEmbeddingId(

internalId = InternalId.TweetId(tweetId),

modelVersion = ModelVersion.Model20m145k2020,

embeddingType = EmbeddingType.LogFavBasedTweet

)

))

)

}

private def getTweetScoreId(

sourceTweetId: TweetId,

candidateTweetId: TweetId

) = {

ScoreId(

algorithm = ScoringAlgorithm.PairEmbeddingCosineSimilarity,

internalId = ScoreInternalId.SimClustersEmbeddingPairScoreId(

SimClustersEmbeddingPairScoreId(

SimClustersEmbeddingId(

internalId = InternalId.TweetId(sourceTweetId),

modelVersion = ModelVersion.Model20m145k2020,

embeddingType = EmbeddingType.LogFavLongestL2EmbeddingTweet

),

SimClustersEmbeddingId(

internalId = InternalId.TweetId(candidateTweetId),

modelVersion = ModelVersion.Model20m145k2020,

embeddingType = EmbeddingType.LogFavBasedTweet

)

))

)

}

}