package com.twitter.simclusters\_v2.scalding.offline\_job

import com.twitter.algebird.Aggregator.size

import com.twitter.algebird.{Aggregator, QTreeAggregatorLowerBound}

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

import com.twitter.simclusters\_v2.candidate\_source.\_

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

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

ClusterTopKTweetsWithScores,

ClustersUserIsInterestedIn

}

import java.nio.ByteBuffer

case class OfflineRecConfig(

maxTweetRecs: Int, // total number of tweet recs.

maxTweetsPerUser: Int,

maxClustersToQuery: Int,

minTweetScoreThreshold: Double,

rankClustersBy: ClusterRanker.Value)

/\*\*

\* An offline simulation of the tweet rec logic in [[InterestedInTweetCandidateStore]].

\* The main difference is that instead of using Memcache, it uses an offline clusterTopK store as

\* the tweet source.

\* Also, instead of taking a single userId as input, it processes a pipe of users altogether.

\*/

object OfflineTweetRecommendation {

case class ScoredTweet(tweetId: TweetId, score: Double) {

def toTuple: (TweetId, Double) = {

(tweetId, score)

}

}

object ScoredTweet {

def apply(tuple: (TweetId, Double)): ScoredTweet = new ScoredTweet(tuple.\_1, tuple.\_2)

implicit val scoredOrdering: Ordering[ScoredTweet] = (x: ScoredTweet, y: ScoredTweet) => {

Ordering.Double.compare(x.score, y.score)

}

}

def getTopTweets(

config: OfflineRecConfig,

targetUsersPipe: TypedPipe[Long],

userIsInterestedInPipe: TypedPipe[(Long, ClustersUserIsInterestedIn)],

clusterTopKTweetsPipe: TypedPipe[ClusterTopKTweetsWithScores]

)(

implicit uniqueID: UniqueID

): Execution[TypedPipe[(Long, Seq[ScoredTweet])]] = {

val tweetRecommendedCount = Stat("NumTweetsRecomended")

val targetUserCount = Stat("NumTargetUsers")

val userWithRecsCount = Stat("NumUsersWithAtLeastTweetRec")

// For every user, read the user's interested-in clusters and cluster's weights

val userClusterWeightPipe: TypedPipe[(Int, (Long, Double))] =

targetUsersPipe.asKeys

.join(userIsInterestedInPipe)

.flatMap {

case (userId, (\_, clustersWithScores)) =>

targetUserCount.inc()

val topClusters = ClusterRanker

.getTopKClustersByScore(

clustersWithScores.clusterIdToScores.toMap,

ClusterRanker.RankByNormalizedFavScore,

config.maxClustersToQuery

).toList

topClusters.map {

case (clusterId, clusterWeightForUser) =>

(clusterId, (userId, clusterWeightForUser))

}

}

// For every cluster, read the top tweets in the cluster, and their weights

val clusterTweetWeightPipe: TypedPipe[(Int, List[(Long, Double)])] =

clusterTopKTweetsPipe

.flatMap { cluster =>

val tweets =

cluster.topKTweets.toList // Convert to a List, otherwise .flatMap dedups by clusterIds

.flatMap {

case (tid, persistedScores) =>

val tweetWeight = persistedScores.score.map(\_.value).getOrElse(0.0)

if (tweetWeight > 0) {

Some((tid, tweetWeight))

} else {

None

}

}

if (tweets.nonEmpty) {

Some((cluster.clusterId, tweets))

} else {

None

}

}

// Collect all the tweets from clusters user is interested in

val recommendedTweetsPipe = userClusterWeightPipe

.sketch(4000)(cid => ByteBuffer.allocate(4).putInt(cid).array(), Ordering.Int)

.join(clusterTweetWeightPipe)

.flatMap {

case (\_, ((userId, clusterWeight), tweetsPerCluster)) =>

tweetsPerCluster.map {

case (tid, tweetWeight) =>

val contribution = clusterWeight \* tweetWeight

((userId, tid), contribution)

}

}

.sumByKey

.withReducers(5000)

// Filter by minimum score threshold

val scoreFilteredTweetsPipe = recommendedTweetsPipe

.collect {

case ((userId, tid), score) if score >= config.minTweetScoreThreshold =>

(userId, ScoredTweet(tid, score))

}

// Rank top tweets for each user

val topTweetsPerUserPipe = scoreFilteredTweetsPipe.group

.sortedReverseTake(config.maxTweetsPerUser)(ScoredTweet.scoredOrdering)

.flatMap {

case (userId, tweets) =>

userWithRecsCount.inc()

tweetRecommendedCount.incBy(tweets.size)

tweets.map { t => (userId, t) }

}

.forceToDiskExecution

val topTweetsPipe = topTweetsPerUserPipe

.flatMap { tweets =>

approximateScoreAtTopK(tweets.map(\_.\_2.score), config.maxTweetRecs).map { threshold =>

tweets

.collect {

case (userId, tweet) if tweet.score >= threshold =>

(userId, List(tweet))

}

.sumByKey

.toTypedPipe

}

}

topTweetsPipe

}

/\*\*

\* Returns the approximate score at the k'th top ranked record using sampling.

\* This score can then be used to filter for the top K elements in a big pipe where

\* K is too big to fit in memory.

\*

\*/

def approximateScoreAtTopK(pipe: TypedPipe[Double], topK: Int): Execution[Double] = {

val defaultScore = 0.0

println("approximateScoreAtTopK: topK=" + topK)

pipe

.aggregate(size)

.getOrElseExecution(0L)

.flatMap { len =>

println("approximateScoreAtTopK: len=" + len)

val topKPercentile = if (len == 0 || topK > len) 0 else 1 - topK.toDouble / len.toDouble

val randomSample = Aggregator.reservoirSample[Double](Math.max(100000, topK / 100))

pipe

.aggregate(randomSample)

.getOrElseExecution(List.empty)

.flatMap { sample =>

TypedPipe

.from(sample)

.aggregate(QTreeAggregatorLowerBound[Double](topKPercentile))

.getOrElseExecution(defaultScore)

}

}

.map { score =>

println("approximateScoreAtTopK: topK percentile score=" + score)

score

}

}

}