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

import com.twitter.scalding.\_

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

import com.twitter.simclusters\_v2.summingbird.common.{Configs, SimClustersInterestedInUtil}

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

import java.util.TimeZone

object SimClustersOfflineJob {

import SimClustersOfflineJobUtil.\_

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

val modelVersionMap: Map[String, PersistedModelVersion] = Map(

ModelVersions.Model20M145KDec11 -> PersistedModelVersion.Model20m145kDec11,

ModelVersions.Model20M145KUpdated -> PersistedModelVersion.Model20m145kUpdated

)

/\*\*

\* Get a list of tweets that received at least one fav in the last tweetTtl Duration

\*/

def getSubsetOfValidTweets(tweetTtl: Duration)(implicit dateRange: DateRange): TypedPipe[Long] = {

readTimelineFavoriteData(DateRange(dateRange.end - tweetTtl, dateRange.end)).map(\_.\_2).distinct

}

/\*\*

\* Note that this job will write several types of scores into the same data set. Please use filter

\* to take the score types you need.

\*/

def computeAggregatedTweetClusterScores(

dateRange: DateRange,

userInterestsData: TypedPipe[(Long, ClustersUserIsInterestedIn)],

favoriteData: TypedPipe[(UserId, TweetId, Timestamp)],

previousTweetClusterScores: TypedPipe[TweetAndClusterScores]

)(

implicit timeZone: TimeZone,

uniqueID: UniqueID

): TypedPipe[TweetAndClusterScores] = {

val latestTimeStamp = dateRange.end.timestamp

val currentScores: TypedPipe[

((Long, Int, PersistedModelVersion, Option[PersistedScoreType]), PersistedScores)

] =

favoriteData

.map {

case (userId, tweetId, timestamp) =>

(userId, (tweetId, timestamp))

}

.count("NumFavEvents")

.leftJoin(userInterestsData)

.withReducers(600)

.flatMap {

case (\_, ((tweetId, timestamp), Some(userInterests))) =>

val clustersWithScores =

SimClustersInterestedInUtil.topClustersWithScores(userInterests)

(

for {

(clusterId, scores) <- clustersWithScores

if scores.favScore >= Configs.favScoreThresholdForUserInterest(

userInterests.knownForModelVersion)

} yield {

// write several types of scores

Seq(

(

tweetId,

clusterId,

modelVersionMap(userInterests.knownForModelVersion),

Some(PersistedScoreType.NormalizedFav8HrHalfLife)) ->

// let the score decay to latestTimeStamp

persistedScoresMonoid.plus(

persistedScoresMonoid

.build(scores.clusterNormalizedFavScore, timestamp),

persistedScoresMonoid.build(0.0, latestTimeStamp)

),

(

tweetId,

clusterId,

modelVersionMap(userInterests.knownForModelVersion),

Some(PersistedScoreType.NormalizedFollow8HrHalfLife)) ->

// let the score decay to latestTimeStamp

persistedScoresMonoid.plus(

persistedScoresMonoid

.build(scores.clusterNormalizedFollowScore, timestamp),

persistedScoresMonoid.build(0.0, latestTimeStamp)

),

(

tweetId,

clusterId,

modelVersionMap(userInterests.knownForModelVersion),

Some(PersistedScoreType.NormalizedLogFav8HrHalfLife)) ->

// let the score decay to latestTimeStamp

persistedScoresMonoid.plus(

persistedScoresMonoid

.build(scores.clusterNormalizedLogFavScore, timestamp),

persistedScoresMonoid.build(0.0, latestTimeStamp)

)

)

}

).flatten

case \_ =>

Nil

}

.count("NumTweetClusterScoreUpdates")

.sumByLocalKeys // there is a .sumByKey later, so just doing a local sum here.

val previousScores: TypedPipe[

((Long, Int, PersistedModelVersion, Option[PersistedScoreType]), PersistedScores)

] =

previousTweetClusterScores.map { v =>

(v.tweetId, v.clusterId, v.modelVersion, v.scoreType) -> v.scores

}

// add current scores and previous scores

(currentScores ++ previousScores).sumByKey

.withReducers(1000)

.map {

case ((tweetId, clusterId, modelVersion, scoreType), scores) =>

TweetAndClusterScores(tweetId, clusterId, modelVersion, scores, scoreType)

}

.count("NumAggregatedTweetClusterScores")

}

def computeTweetTopKClusters(

latestTweetClusterScores: TypedPipe[TweetAndClusterScores],

topK: Int = Configs.topKClustersPerTweet,

scoreThreshold: Double = Configs.scoreThresholdForEntityTopKClustersCache

)(

implicit timeZone: TimeZone,

uniqueID: UniqueID

): TypedPipe[TweetTopKClustersWithScores] = {

latestTweetClusterScores

.flatMap { v =>

val score = v.scores.score.map(\_.value).getOrElse(0.0)

if (score < scoreThreshold) {

None

} else {

Some((v.tweetId, v.modelVersion, v.scoreType) -> (v.clusterId, v.scores))

}

}

.count("NumAggregatedTweetClusterScoresAfterFilteringInTweetTopK")

.group

.sortedReverseTake(topK)(Ordering.by(\_.\_2))

.map {

case ((tweetId, modelVersion, scoreType), topKClusters) =>

TweetTopKClustersWithScores(tweetId, modelVersion, topKClusters.toMap, scoreType)

}

.count("NumTweetTopK")

}

def computeClusterTopKTweets(

latestTweetClusterScores: TypedPipe[TweetAndClusterScores],

topK: Int = Configs.topKTweetsPerCluster,

scoreThreshold: Double = Configs.scoreThresholdForClusterTopKTweetsCache

)(

implicit timeZone: TimeZone,

uniqueID: UniqueID

): TypedPipe[ClusterTopKTweetsWithScores] = {

latestTweetClusterScores

.flatMap { v =>

val score = v.scores.score.map(\_.value).getOrElse(0.0)

if (score < scoreThreshold) {

None

} else {

Some((v.clusterId, v.modelVersion, v.scoreType) -> (v.tweetId, v.scores))

}

}

.count("NumAggregatedTweetClusterScoresAfterFilteringInClusterTopK")

.group

.sortedReverseTake(topK)(Ordering.by(\_.\_2))

.map {

case ((clusterId, modelVersion, scoreType), topKTweets) =>

ClusterTopKTweetsWithScores(clusterId, modelVersion, topKTweets.toMap, scoreType)

}

.count("NumClusterTopK")

}

}