package com.twitter.timelines.data\_processing.ad\_hoc.earlybird\_ranking.model\_evaluation

import com.twitter.algebird.Aggregator

import com.twitter.algebird.AveragedValue

import com.twitter.ml.api.prediction\_engine.PredictionEnginePlugin

import com.twitter.ml.api.util.FDsl

import com.twitter.ml.api.DataRecord

import com.twitter.ml.api.IRecordOneToManyAdapter

import com.twitter.scalding.Args

import com.twitter.scalding.DateRange

import com.twitter.scalding.Execution

import com.twitter.scalding.TypedJson

import com.twitter.scalding.TypedPipe

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

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

import com.twitter.timelines.data\_processing.ad\_hoc.earlybird\_ranking.common.EarlybirdTrainingRecapConfiguration

import com.twitter.timelines.data\_processing.util.RequestImplicits.RichRequest

import com.twitter.timelines.data\_processing.util.example.RecapTweetExample

import com.twitter.timelines.data\_processing.util.execution.UTCDateRangeFromArgs

import com.twitter.timelines.prediction.adapters.recap.RecapSuggestionRecordAdapter

import com.twitter.timelines.prediction.features.recap.RecapFeatures

import com.twitter.timelines.suggests.common.record.thriftscala.SuggestionRecord

import com.twitter.timelineservice.suggests.logging.recap.thriftscala.HighlightTweet

import com.twitter.timelineservice.suggests.logging.thriftscala.SuggestsRequestLog

import scala.collection.JavaConverters.\_

import scala.language.reflectiveCalls

import scala.util.Random

import twadoop\_config.configuration.log\_categories.group.timelines.TimelineserviceInjectionRequestLogScalaDataset

/\*\*

\* Evaluates an Earlybird model using 1% injection request logs.

\*

\* Arguments:

\* --model\_base\_path path to Earlybird model snapshots

\* --models list of model names to evaluate

\* --output path to output stats

\* --parallelism (default: 3) number of tasks to run in parallel

\* --topks (optional) list of values of `k` (integers) for top-K metrics

\* --topn\_fractions (optional) list of values of `n` (doubles) for top-N-fraction metrics

\* --seed (optional) seed for random number generator

\*/

object EarlybirdModelEvaluationJob extends TwitterExecutionApp with UTCDateRangeFromArgs {

import FDsl.\_

import PredictionEnginePlugin.\_

private[this] val averager: Aggregator[Double, AveragedValue, Double] =

AveragedValue.aggregator

private[this] val recapAdapter: IRecordOneToManyAdapter[SuggestionRecord] =

new RecapSuggestionRecordAdapter(checkDwellTime = false)

override def job: Execution[Unit] = {

for {

args <- Execution.getArgs

dateRange <- dateRangeEx

metrics = getMetrics(args)

random = buildRandom(args)

modelBasePath = args("model\_base\_path")

models = args.list("models")

parallelism = args.int("parallelism", 3)

logs = logsHavingCandidates(dateRange)

modelScoredCandidates = models.map { model =>

(model, scoreCandidatesUsingModel(logs, s"$modelBasePath/$model"))

}

functionScoredCandidates = List(

("random", scoreCandidatesUsingFunction(logs, \_ => Some(random.nextDouble()))),

("original\_earlybird", scoreCandidatesUsingFunction(logs, extractOriginalEarlybirdScore)),

("blender", scoreCandidatesUsingFunction(logs, extractBlenderScore))

)

allCandidates = modelScoredCandidates ++ functionScoredCandidates

statsExecutions = allCandidates.map {

case (name, pipe) =>

for {

saved <- pipe.forceToDiskExecution

stats <- computeMetrics(saved, metrics, parallelism)

} yield (name, stats)

}

stats <- Execution.withParallelism(statsExecutions, parallelism)

\_ <- TypedPipe.from(stats).writeExecution(TypedJson(args("output")))

} yield ()

}

private[this] def computeMetrics(

requests: TypedPipe[Seq[CandidateRecord]],

metricsToCompute: Seq[EarlybirdEvaluationMetric],

parallelism: Int

): Execution[Map[String, Double]] = {

val metricExecutions = metricsToCompute.map { metric =>

val metricEx = requests.flatMap(metric(\_)).aggregate(averager).toOptionExecution

metricEx.map { value => value.map((metric.name, \_)) }

}

Execution.withParallelism(metricExecutions, parallelism).map(\_.flatten.toMap)

}

private[this] def getMetrics(args: Args): Seq[EarlybirdEvaluationMetric] = {

val topKs = args.list("topks").map(\_.toInt)

val topNFractions = args.list("topn\_fractions").map(\_.toDouble)

val topKMetrics = topKs.flatMap { topK =>

Seq(

TopKRecall(topK, filterFewerThanK = false),

TopKRecall(topK, filterFewerThanK = true),

ShownTweetRecall(topK),

AverageFullScoreForTopLight(topK),

SumScoreRecallForTopLight(topK),

HasFewerThanKCandidates(topK),

ShownTweetRecallWithFullScores(topK),

ProbabilityOfCorrectOrdering

)

}

val topNPercentMetrics = topNFractions.flatMap { topNPercent =>

Seq(

TopNPercentRecall(topNPercent),

ShownTweetPercentRecall(topNPercent)

)

}

topKMetrics ++ topNPercentMetrics ++ Seq(NumberOfCandidates)

}

private[this] def buildRandom(args: Args): Random = {

val seedOpt = args.optional("seed").map(\_.toLong)

seedOpt.map(new Random(\_)).getOrElse(new Random())

}

private[this] def logsHavingCandidates(dateRange: DateRange): TypedPipe[SuggestsRequestLog] =

DAL

.read(TimelineserviceInjectionRequestLogScalaDataset, dateRange)

.toTypedPipe

.filter(\_.recapCandidates.exists(\_.nonEmpty))

/\*\*

\* Uses a model defined at `earlybirdModelPath` to score candidates and

\* returns a Seq[CandidateRecord] for each request.

\*/

private[this] def scoreCandidatesUsingModel(

logs: TypedPipe[SuggestsRequestLog],

earlybirdModelPath: String

): TypedPipe[Seq[CandidateRecord]] = {

logs

.usingScorer(earlybirdModelPath)

.map {

case (scorer: PredictionEngineScorer, log: SuggestsRequestLog) =>

val suggestionRecords =

RecapTweetExample

.extractCandidateTweetExamples(log)

.map(\_.asSuggestionRecord)

val servedTweetIds = log.servedHighlightTweets.flatMap(\_.tweetId).toSet

val renamer = (new EarlybirdTrainingRecapConfiguration).EarlybirdFeatureRenamer

suggestionRecords.flatMap { suggestionRecord =>

val dataRecordOpt = recapAdapter.adaptToDataRecords(suggestionRecord).asScala.headOption

dataRecordOpt.foreach(renamer.transform)

for {

tweetId <- suggestionRecord.itemId

fullScore <- suggestionRecord.recapFeatures.flatMap(\_.combinedModelScore)

earlybirdScore <- dataRecordOpt.flatMap(calculateLightScore(\_, scorer))

} yield CandidateRecord(

tweetId = tweetId,

fullScore = fullScore,

earlyScore = earlybirdScore,

served = servedTweetIds.contains(tweetId)

)

}

}

}

/\*\*

\* Uses a simple function to score candidates and returns a Seq[CandidateRecord] for each

\* request.

\*/

private[this] def scoreCandidatesUsingFunction(

logs: TypedPipe[SuggestsRequestLog],

earlyScoreExtractor: HighlightTweet => Option[Double]

): TypedPipe[Seq[CandidateRecord]] = {

logs

.map { log =>

val tweetCandidates = log.recapTweetCandidates.getOrElse(Nil)

val servedTweetIds = log.servedHighlightTweets.flatMap(\_.tweetId).toSet

for {

candidate <- tweetCandidates

tweetId <- candidate.tweetId

fullScore <- candidate.recapFeatures.flatMap(\_.combinedModelScore)

earlyScore <- earlyScoreExtractor(candidate)

} yield CandidateRecord(

tweetId = tweetId,

fullScore = fullScore,

earlyScore = earlyScore,

served = servedTweetIds.contains(tweetId)

)

}

}

private[this] def extractOriginalEarlybirdScore(candidate: HighlightTweet): Option[Double] =

for {

recapFeatures <- candidate.recapFeatures

tweetFeatures <- recapFeatures.tweetFeatures

} yield tweetFeatures.earlybirdScore

private[this] def extractBlenderScore(candidate: HighlightTweet): Option[Double] =

for {

recapFeatures <- candidate.recapFeatures

tweetFeatures <- recapFeatures.tweetFeatures

} yield tweetFeatures.blenderScore

private[this] def calculateLightScore(

dataRecord: DataRecord,

scorer: PredictionEngineScorer

): Option[Double] = {

val scoredRecord = scorer(dataRecord)

if (scoredRecord.hasFeature(RecapFeatures.PREDICTED\_IS\_UNIFIED\_ENGAGEMENT)) {

Some(scoredRecord.getFeatureValue(RecapFeatures.PREDICTED\_IS\_UNIFIED\_ENGAGEMENT).toDouble)

} else {

None

}

}

}