package com.twitter.simclusters\_v2.scalding.tweet\_similarity

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

import com.twitter.ml.api.{DataRecord, DataRecordMerger, DataSetPipe, FeatureContext}

import com.twitter.ml.featurestore.lib.data.EntityIds.Entry

import com.twitter.ml.featurestore.lib.data.{EntityIds, FeatureValuesById, PredictionRecord}

import com.twitter.scalding.typed.TypedPipe

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

import com.twitter.simclusters\_v2.tweet\_similarity.ModelBasedTweetSimilaritySimClustersEmbeddingAdapter.{

NormalizedCandidateEmbAdapter,

NormalizedQueryEmbAdapter

}

import com.twitter.simclusters\_v2.tweet\_similarity.{

TweetSimilarityFeatures,

TweetSimilarityFeaturesStoreConfig

}

import com.twitter.simclusters\_v2.common.{Timestamp, TweetId, UserId}

import com.twitter.simclusters\_v2.scalding.tweet\_similarity.TweetPairLabelCollectionUtil.FeaturedTweet

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

PersistentSimClustersEmbedding,

SimClustersEmbedding => ThriftSimClustersEmbedding

}

object TweetPairFeatureHydrationUtil {

val QueryTweetConfig = new TweetSimilarityFeaturesStoreConfig("query\_tweet\_user\_id")

val CandidateTweetConfig = new TweetSimilarityFeaturesStoreConfig("candidate\_tweet\_user\_id")

val DataRecordMerger = new DataRecordMerger()

/\*\*

\* Given persistentEmbeddings TypedPipe, extract tweetId, timestamp, and the embedding

\*

\* @param persistentEmbeddings TypedPipe of ((TweetId, Timestamp), PersistentSimClustersEmbedding), read from PersistentTweetEmbeddingMhExportSource

\*

\* @return Extracted TypedPipe of (TweetId, (Timestamp, SimClustersEmbedding))

\*/

def extractEmbeddings(

persistentEmbeddings: TypedPipe[((TweetId, Timestamp), PersistentSimClustersEmbedding)]

): TypedPipe[(TweetId, (Timestamp, ThriftSimClustersEmbedding))] = {

persistentEmbeddings

.collect {

case ((tweetId, \_), embedding) if embedding.metadata.updatedAtMs.isDefined =>

(tweetId, (embedding.metadata.updatedAtMs.get, embedding.embedding))

}

}

/\*\*

\* Hydrate the tweet pairs with the latest persistent embeddings before engagement/impression.

\*

\* @param tweetPairs TypedPipe of the (userId, queryFeaturedTweet, candidateFeaturedTweet, label)

\* @param persistentEmbeddings TypedPipe of persistentEmbeddings from PersistentTweetEmbeddingMhExportSource

\*

\* @return TypedPipe of the (userId, queryFeaturedTweet, candidateFeaturedTweet, label) with persistent embeddings set

\*/

def getTweetPairsWithPersistentEmbeddings(

tweetPairs: TypedPipe[(FeaturedTweet, FeaturedTweet, Boolean)],

persistentEmbeddings: TypedPipe[((TweetId, Timestamp), PersistentSimClustersEmbedding)]

): TypedPipe[(FeaturedTweet, FeaturedTweet, Boolean)] = {

val extractedEmbeddings = extractEmbeddings(persistentEmbeddings)

tweetPairs

.groupBy {

case (queryFeaturedTweet, \_, \_) => queryFeaturedTweet.tweet

}

.join(extractedEmbeddings)

.collect {

case (

\_,

(

(queryFeaturedTweet, candidateFeaturedTweet, label),

(embeddingTimestamp, embedding)))

if embeddingTimestamp <= queryFeaturedTweet.timestamp =>

((queryFeaturedTweet, candidateFeaturedTweet), (embeddingTimestamp, embedding, label))

}

.group

.maxBy(\_.\_1)

.map {

case ((queryFeaturedTweet, candidateFeaturedTweet), (\_, embedding, label)) =>

(

candidateFeaturedTweet.tweet,

(queryFeaturedTweet.copy(embedding = Some(embedding)), candidateFeaturedTweet, label)

)

}

.join(extractedEmbeddings)

.collect {

case (

\_,

(

(queryFeaturedTweet, candidateFeaturedTweet, label),

(embeddingTimestamp, embedding)))

if embeddingTimestamp <= candidateFeaturedTweet.timestamp =>

((queryFeaturedTweet, candidateFeaturedTweet), (embeddingTimestamp, embedding, label))

}

.group

.maxBy(\_.\_1)

.map {

case ((queryFeaturedTweet, candidateFeaturedTweet), (\_, embedding, label)) =>

(queryFeaturedTweet, candidateFeaturedTweet.copy(embedding = Some(embedding)), label)

}

}

/\*\*

\* Get tweet pairs with the author userIds

\*

\* @param tweetPairs TypedPipe of (queryTweet, queryEmbedding, queryTimestamp, candidateTweet, candidateEmbedding, candidateTimestamp, label)

\* @param tweetAuthorPairs TypedPipe of (tweetId, author userId)

\*

\* @return TypedPipe of (queryTweet, queryAuthor, queryEmbedding, queryTimestamp, candidateTweet, candidateAuthor, candidateEmbedding, candidateTimestamp, label)

\*/

def getTweetPairsWithAuthors(

tweetPairs: TypedPipe[(FeaturedTweet, FeaturedTweet, Boolean)],

tweetAuthorPairs: TypedPipe[(TweetId, UserId)]

): TypedPipe[(FeaturedTweet, FeaturedTweet, Boolean)] = {

tweetPairs

//keyed by queryTweet s.t. we get queryTweet's author after joining with tweetAuthorPairs

.groupBy { case (queryFeaturedTweet, \_, \_) => queryFeaturedTweet.tweet }

.join(tweetAuthorPairs)

.values

//keyed by candidateTweet

.groupBy { case ((\_, candidateFeaturedTweet, \_), \_) => candidateFeaturedTweet.tweet }

.join(tweetAuthorPairs)

.values

.map {

case (

((queryFeaturedTweet, candidateFeaturedTweet, label), queryAuthor),

candidateAuthor) =>

(

queryFeaturedTweet.copy(author = Some(queryAuthor)),

candidateFeaturedTweet.copy(author = Some(candidateAuthor)),

label

)

}

}

/\*\*

\* Get tweet pairs with popularity counts

\*

\* @param tweetPairs TypedPipe of the (userId, queryFeaturedTweet, candidateFeaturedTweet, label)

\*

\* @return TypedPipe of the (userId, queryFeaturedTweet, candidateFeaturedTweet, tweetPairCount, queryTweetCount, label)

\*/

def getTweetPairsWithCounts(

tweetPairs: TypedPipe[(FeaturedTweet, FeaturedTweet, Boolean)]

): TypedPipe[(FeaturedTweet, FeaturedTweet, Long, Long, Boolean)] = {

val tweetPairCount = tweetPairs.groupBy {

case (queryFeaturedTweet, candidateFeaturedTweet, \_) =>

(queryFeaturedTweet.tweet, candidateFeaturedTweet.tweet)

}.size

val queryTweetCount = tweetPairs.groupBy {

case (queryFeaturedTweet, \_, \_) => queryFeaturedTweet.tweet

}.size

tweetPairs

.groupBy {

case (queryFeaturedTweet, candidateFeaturedTweet, \_) =>

(queryFeaturedTweet.tweet, candidateFeaturedTweet.tweet)

}

.join(tweetPairCount)

.values

.map {

case ((queryFeaturedTweet, candidateFeaturedTweet, label), tweetPairCount) =>

(queryFeaturedTweet, candidateFeaturedTweet, tweetPairCount, label)

}

.groupBy { case (queryFeaturedTweet, \_, \_, \_) => queryFeaturedTweet.tweet }

.join(queryTweetCount)

.values

.map {

case (

(queryFeaturedTweet, candidateFeaturedTweet, tweetPairCount, label),

queryTweetCount) =>

(queryFeaturedTweet, candidateFeaturedTweet, tweetPairCount, queryTweetCount, label)

}

}

/\*\*

\* Get training data records

\*

\* @param tweetPairs TypedPipe of the (userId, queryFeaturedTweet, candidateFeaturedTweet, label)

\* @param persistentEmbeddings TypedPipe of persistentEmbeddings from PersistentTweetEmbeddingMhExportSource

\* @param tweetAuthorPairs TypedPipe of (tweetId, author userId)

\* @param useAuthorFeatures whether to use author features or not

\*

\* @return DataSetPipe with features and label

\*/

def getDataSetPipeWithFeatures(

tweetPairs: TypedPipe[(FeaturedTweet, FeaturedTweet, Boolean)],

persistentEmbeddings: TypedPipe[((TweetId, Timestamp), PersistentSimClustersEmbedding)],

tweetAuthorPairs: TypedPipe[(TweetId, UserId)],

useAuthorFeatures: Boolean

): DataSetPipe = {

val featuredTweetPairs =

if (useAuthorFeatures)

getTweetPairsWithCounts(

getTweetPairsWithPersistentEmbeddings(

getTweetPairsWithAuthors(tweetPairs, tweetAuthorPairs),

persistentEmbeddings))

else

getTweetPairsWithCounts(

getTweetPairsWithPersistentEmbeddings(tweetPairs, persistentEmbeddings))

DataSetPipe(

featuredTweetPairs.flatMap {

case (queryFeaturedTweet, candidateFeaturedTweet, tweetPairCount, queryTweetCount, label) =>

getDataRecordWithFeatures(

queryFeaturedTweet,

candidateFeaturedTweet,

tweetPairCount,

queryTweetCount,

label)

},

FeatureContext.merge(

TweetSimilarityFeatures.FeatureContext,

QueryTweetConfig.predictionRecordAdapter.getFeatureContext,

CandidateTweetConfig.predictionRecordAdapter.getFeatureContext

)

)

}

/\*\*

\* Given raw features, return a DataRecord with all the features

\*

\* @param queryFeaturedTweet FeaturedTweet for query tweet

\* @param candidateFeaturedTweet FeaturedTweet for candidate tweet

\* @param tweetPairCount popularity count for the (query tweet, candidate tweet) pair

\* @param queryTweetCount popularity count for each query tweet

\* @param label true for positive and false for negative

\*

\* @return

\*/

def getDataRecordWithFeatures(

queryFeaturedTweet: FeaturedTweet,

candidateFeaturedTweet: FeaturedTweet,

tweetPairCount: Long,

queryTweetCount: Long,

label: Boolean

): Option[DataRecord] = {

for {

queryEmbedding <- queryFeaturedTweet.embedding

candidateEmbedding <- candidateFeaturedTweet.embedding

} yield {

val featureDataRecord = NormalizedQueryEmbAdapter.adaptToDataRecord(queryEmbedding)

DataRecordMerger.merge(

featureDataRecord,

NormalizedCandidateEmbAdapter.adaptToDataRecord(candidateEmbedding))

featureDataRecord.setFeatureValue(

TweetSimilarityFeatures.QueryTweetId,

queryFeaturedTweet.tweet)

featureDataRecord.setFeatureValue(

TweetSimilarityFeatures.CandidateTweetId,

candidateFeaturedTweet.tweet)

featureDataRecord.setFeatureValue(

TweetSimilarityFeatures.QueryTweetTimestamp,

queryFeaturedTweet.timestamp)

featureDataRecord.setFeatureValue(

TweetSimilarityFeatures.CandidateTweetTimestamp,

candidateFeaturedTweet.timestamp)

featureDataRecord.setFeatureValue(

TweetSimilarityFeatures.CosineSimilarity,

queryEmbedding.cosineSimilarity(candidateEmbedding))

featureDataRecord.setFeatureValue(TweetSimilarityFeatures.TweetPairCount, tweetPairCount)

featureDataRecord.setFeatureValue(TweetSimilarityFeatures.QueryTweetCount, queryTweetCount)

featureDataRecord.setFeatureValue(TweetSimilarityFeatures.Label, label)

if (queryFeaturedTweet.author.isDefined && candidateFeaturedTweet.author.isDefined) {

DataRecordMerger.merge(

featureDataRecord,

new DataRecord(

QueryTweetConfig.predictionRecordAdapter.adaptToDataRecord(PredictionRecord(

FeatureValuesById.empty,

EntityIds(Entry(

QueryTweetConfig.bindingIdentifier,

Set(com.twitter.ml.featurestore.lib.UserId(queryFeaturedTweet.author.get))))

)))

)

DataRecordMerger.merge(

featureDataRecord,

new DataRecord(

CandidateTweetConfig.predictionRecordAdapter.adaptToDataRecord(PredictionRecord(

FeatureValuesById.empty,

EntityIds(Entry(

CandidateTweetConfig.bindingIdentifier,

Set(com.twitter.ml.featurestore.lib.UserId(candidateFeaturedTweet.author.get))))

)))

)

}

featureDataRecord

}

}

}