package com.twitter.search.common.util.ml;

import java.util.List;

import java.util.Map;

import java.util.Optional;

import com.google.common.base.Preconditions;

import com.google.common.collect.Sets;

/\*\*

\* Utilities for feature transformation and extraction.

\*/

public final class FeatureUtils {

private FeatureUtils() {

}

/\*\*

\* Computes the difference between 2 values and returns the ratio of the difference over the

\* minimum of both, according to these cases:

\*

\* 1. if (a > b) return a / b

\* 2. if (a < b) return - b / a

\* 3. if (a == b == 0) return 0

\*

\* The upper/lower limit is (-) maxRatio. For cases 1 and 2, if the denominator is 0,

\* it returns maxRatio.

\*

\* This method is used to define a feature that tells how much larger or smaller is the

\* first value with respect to the second one..

\*/

public static float diffRatio(float a, float b, float maxRatio) {

float diff = a - b;

if (diff == 0) {

return 0;

}

float denominator = Math.min(a, b);

float ratio = denominator != 0 ? Math.abs(diff / denominator) : maxRatio;

return Math.copySign(Math.min(ratio, maxRatio), diff);

}

/\*\*

\* Computes the cosine similarity between two maps that represent sparse vectors.

\*/

public static <K, V extends Number> double cosineSimilarity(

Map<K, V> vector1, Map<K, V> vector2) {

if (vector1 == null || vector1.isEmpty() || vector2 == null || vector2.isEmpty()) {

return 0;

}

double squaredSum1 = 0;

double squaredSum2 = 0;

double squaredCrossSum = 0;

for (K key : Sets.union(vector1.keySet(), vector2.keySet())) {

double value1 = 0;

double value2 = 0;

V optValue1 = vector1.get(key);

if (optValue1 != null) {

value1 = optValue1.doubleValue();

}

V optValue2 = vector2.get(key);

if (optValue2 != null) {

value2 = optValue2.doubleValue();

}

squaredSum1 += value1 \* value1;

squaredSum2 += value2 \* value2;

squaredCrossSum += value1 \* value2;

}

if (squaredSum1 == 0 || squaredSum2 == 0) {

return 0;

} else {

return squaredCrossSum / Math.sqrt(squaredSum1 \* squaredSum2);

}

}

/\*\*

\* Computes the cosine similarity between two (dense) vectors.

\*/

public static <V extends Number> double cosineSimilarity(

List<V> vector1, List<V> vector2) {

if (vector1 == null || vector1.isEmpty() || vector2 == null || vector2.isEmpty()) {

return 0;

}

Preconditions.checkArgument(vector1.size() == vector2.size());

double squaredSum1 = 0;

double squaredSum2 = 0;

double squaredCrossSum = 0;

for (int i = 0; i < vector1.size(); i++) {

double value1 = vector1.get(i).doubleValue();

double value2 = vector2.get(i).doubleValue();

squaredSum1 += value1 \* value1;

squaredSum2 += value2 \* value2;

squaredCrossSum += value1 \* value2;

}

if (squaredSum1 == 0 || squaredSum2 == 0) {

return 0;

} else {

return squaredCrossSum / Math.sqrt(squaredSum1 \* squaredSum2);

}

}

/\*\*

\* Finds the key of the map with the highest value (compared in natural order)

\*/

@SuppressWarnings("unchecked")

public static <K, V extends Comparable> Optional<K> findMaxKey(Map<K, V> map) {

if (map == null || map.isEmpty()) {

return Optional.empty();

}

Optional<Map.Entry<K, V>> maxEntry = map.entrySet().stream().max(Map.Entry.comparingByValue());

return maxEntry.map(Map.Entry::getKey);

}

}