Current scope: all classes | org.apache.commons.math4.neuralnet

Coverage Summary for Class: MapRanking (org.apache.commons.math4.neuralnet)

Class	Method, %	Line, %
MapRanking	100% (3/3)	93.5% (29/31)
MapRanking$PairNeuronDouble	100% (4/4)	100% (6/6)
MapRanking$PairNeuronDouble$1	100% (2/2)	100% (2/2)
Total	100% (9/9)	94.9% (37/39)

```java
/*
 * Licensed to the Apache Software Foundation (ASF) under one or more
 * contributor license agreements.  See the NOTICE file distributed with
 * this work for additional information regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License, Version 2.0
 * (the "License"); you may not use this file except in compliance with
 * the License.  You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.apache.commons.math4.neuralnet;

import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

import org.apache.commons.math4.neuralnet.internal.NeuralNetException;

/**
 * Utility for ranking the units (neurons) of a network.
 *
 * @since 4.0
 */
public class MapRanking {
    /** List corresponding to the map passed to the constructor. */
    private final List<Neuron> map = new ArrayList<>();
    /** Distance function for sorting. */
    private final DistanceMeasure distance;

    /**
     * @param neurons List to be ranked.
     * No defensive copy is performed.
     * The {@link #rank(double[],int) created list of units} will
     * be sorted in increasing order of the {@code distance}.
     * @param distance Distance function.
     */
    public MapRanking(Iterable<Neuron> neurons,
                      DistanceMeasure distance) {
        this.distance = distance;

        for (final Neuron n : neurons) {
            map.add(n); // No defensive copy.
        }
    }

    /**
     * Creates a list of the neurons whose features best correspond to the
     * given {@code features}.
     *
     * @param features Data.
     * @return the list of neurons sorted in decreasing order of distance to
     * the given data.
     * @throws IllegalArgumentException if the size of the input is not
     * compatible with the neurons features size.
     */
    public List<Neuron> rank(double[] features) {
        return rank(features, map.size());
    }

    /**
     * Creates a list of the neurons whose features best correspond to the
     * given {@code features}.
     *
     * @param features Data.
     * @param max Maximum size of the returned list.
     * @return the list of neurons sorted in decreasing order of distance to
     * the given data.
     * @throws IllegalArgumentException if the size of the input is not
     * compatible with the neurons features size or {@code max <= 0}.
     */
    public List<Neuron> rank(double[] features,
                             int max) {
        if (max <= 0) {
            throw new NeuralNetException(NeuralNetException.NOT_STRICTLY_POSITIVE, max);
        }
        final int m = max <= map.size() ?
            max :
            map.size();
        final List<PairNeuronDouble> list = new ArrayList<>(m);

        for (final Neuron n : map) {
            final double d = distance.applyAsDouble(n.getFeatures(), features);
            final PairNeuronDouble p = new PairNeuronDouble(n, d);

            if (list.size() < m) {
                list.add(p);
                if (list.size() > 1) {
                    // Sort if there is more than 1 element.
                    Collections.sort(list, PairNeuronDouble.COMPARATOR);
                }
            } else {
                final int last = list.size() - 1;
                if (PairNeuronDouble.COMPARATOR.compare(p, list.get(last)) < 0) {
                    list.set(last, p); // Replace worst entry.
                    if (last > 0) {
                        // Sort if there is more than 1 element.
                        Collections.sort(list, PairNeuronDouble.COMPARATOR);
                    }
                }
            }
        }

        final List<Neuron> result = new ArrayList<>(m);
        for (final PairNeuronDouble p : list) {
            result.add(p.getNeuron());
        }

        return result;
    }

    /**
     * Helper data structure holding a (Neuron, double) pair.
     */
    private static class PairNeuronDouble {
        /** Comparator. */
        static final Comparator<PairNeuronDouble> COMPARATOR
            = new Comparator<PairNeuronDouble>() {
                /** {@inheritDoc} */
                @Override
                public int compare(PairNeuronDouble o1,
                                   PairNeuronDouble o2) {
                    return Double.compare(o1.value, o2.value);
                }
            };
        /** Key. */
        private final Neuron neuron;
        /** Value. */
        private final double value;

```

```java
        /**
         * @param neuron Neuron.
         * @param value Value.
         */
        PairNeuronDouble(Neuron neuron, double value) {
            this.neuron = neuron;
            this.value = value;
        }

        /** @return the neuron. */
        public Neuron getNeuron() {
            return neuron;
        }
    }
}
```

generated on 2023-11-10 12:15