```java
// Appendix A
//
// MovieExplorer: Building an Interactive Exploration Tool from Ratings and Latent Taste Spaces
// Taavi Taijala
// Martijn Willemsen
// Joseph Konstan

import org.apache.commons.math3.linear.RealVector;

import java.util.*;
import java.util.stream.Collectors;

public class Item {
    RealVector vector;
    double score;

    public RealVector getVector() { return vector; }
    public RealVector getDeltaVector(RealVector locationVector) {
        return vector.copy()
                // Scale item vector to lie on the plane perpendicular to the location vector
                .mapDivide(vector.dotProduct(locationVector))
                // Subtract the location vector to get a delta vector
                .subtract(locationVector);
    }

    public void setScore(double s) { score = s; }
    public double getScore() { return score; }
}

public class Utilities {
    Random random;

    public RealVector updateLocationVectorWithFeedback(RealVector locationVector,
                                                       Map<Integer, List<Item>> feedback,
                                                       Integer roundNum) {

        RealVector unitLocationVector = locationVector.unitVector();

        DoubleSummaryStatistics allDVecNorms = feedback.values().stream()
                .flatMap(l -> l.stream())
                .map(i -> i.getDeltaVector(unitLocationVector))
                .mapToDouble(v -> v.getNorm())
                .filter(n -> n > 0)
                .summaryStatistics();
        double totalNorm = allDVecNorms.getSum();
        double maxNorm = allDVecNorms.getMin();
        double minNorm = allDVecNorms.getMax();

        // We update the location vector by adding a direction * distance * confidence to it.
        // We use the following temporary variables to calculate these three values.
        double distanceNumerator = 0.0;
        double distanceDenominator = 0.0;
        RealVector directionNumerator = unitLocationVector.copy().mapMultiply(0.0); // Zero vector
        double directionDenominator = 0.0;
        double selectedNorm = 0.0;

        for (Map.Entry<Integer, List<Item>> entry : feedback.entrySet()) {
            double weight;
            if (entry.getKey() == 1) { // Positive feedback
                weight = Math.sqrt(entry.getValue().size());
            } else if (entry.getKey() == -1) { // Negative feedback
                // It's harder to interpret negative feedback, so we give it half the weight of
                // positive feedback.
                weight = -0.5 * Math.sqrt(entry.getValue().size());
            } else { // Neutral feedback
                continue;
            }

            // Get the delta vectors and the sum of their magnitudes for the given items
            List<RealVector> dVecs = entry.getValue().stream()
```

```
                .map(i -> i.getDeltaVector(unitLocationVector))
                .collect(Collectors.toList());
        Double dVecsNormSum = dVecs.stream().mapToDouble(v -> v.getNorm()).sum();

        // DIRECTION CALCULATIONS //
        // Weight and sum the delta vectors
        RealVector weightedDVec = dVecs.stream()
                .map(v -> v.mapMultiply(weight))
                .reduce((v1, v2) -> v1.add(v2))
                .get();
        directionNumerator = directionNumerator.add(weightedDVec);

        // Add the absolute value of the preferenceWeights to the denominator
        directionDenominator += dVecsNormSum * Math.abs(weight);

        // DISTANCE CALCULATIONS
        if (weight > 0.0) {
            // For positive items, calculate distance as the average vector magnitude
            distanceNumerator += dVecsNormSum;
            distanceDenominator += dVecs.size();
        } else if (weight < 0.0) {
            // For negative items, it doesn't make sense to calculate distnace as the average
            // vector magnitude, since closer negative items with a smaller magnitude should
            // push us farther away than distance negative items with a larger magnitude.

            // When minNorm is close to maxNorm, all item vectors have similar magnitudes and
            // the weight given to negative item vectors will be close to 1. When minNorm is
            // much smaller than maxNorm, item vectors have a range of magnitudes and the weight
            // given to distant negative item vectors will be smaller than the weight given to
            // nearby negative item vectors.
            double totalNegWeight = dVecs.stream()
                    .map(v -> v.getNorm())
                    .mapToDouble(n -> Math.pow(0.1, (n - minNorm) / (2 * minNorm)))
                    .sum();

            // It's harder to interpret negative feedback, so we give it half the weight of
            // positive feedback.
            distanceNumerator += 0.5 * maxNorm * totalNegWeight;
            distanceDenominator += 0.5 * totalNegWeight;
        }

        // CONFIDENCE CALCULATIONS
        selectedNorm += dVecsNormSum;
    }

    RealVector direction = directionNumerator.mapDivide(directionDenominator);
    double distance = distanceNumerator / distanceDenominator;
    double confidence = (1.0 / roundNum)
            + Math.pow(selectedNorm / totalNorm, 1.0 / 3.0)
            * (roundNum - 1.0) / roundNum;

    return unitLocationVector.add(direction.mapMultiply(distance * confidence)).unitVector();
}

public List<Item> getItemsSurroundingLocationVector(RealVector locationVector,
                                                    List<Item> universe,
                                                    List<Item> ratedItems,
                                                    List<Item> previouslyShownItems,
                                                    Integer roundNum) {

    // Exclude items shown in previous rounds to promote novel item discovery
    universe = universe.stream()
            .filter(i -> !previouslyShownItems.contains(i))
            .collect(Collectors.toList());

    List<Item> selected = new ArrayList<>();

    if (roundNum == 1) {
        universe = scoreFilterAndSortItems(locationVector, universe, "wide");
        selectDiverseItems(selected, universe, ratedItems, 5, 100, 0.0, 0.5);
```

```java
            selectDiverseItems(selected, universe, ratedItems, 5, 200, 0.0, 0.5);
        }
        if (roundNum == 2) {
            universe = scoreFilterAndSortItems(locationVector, universe, "medium");
            selectDiverseItems(selected, universe, ratedItems, 3, 5, 0.5, 1.0);
            selectDiverseItems(selected, universe, ratedItems, 3, 40, 0.5, 1.0);
            selectDiverseItems(selected, universe, ratedItems, 4, 100, 0.5, 0.5);
        } else if (roundNum > 2) {
            universe = scoreFilterAndSortItems(locationVector, universe, "narrow");
            selectDiverseItems(selected, universe, ratedItems, 3, 5, 0.5, 1.0);
            selectDiverseItems(selected, universe, ratedItems, 3, 20, 0.5, 1.0);
            selectDiverseItems(selected, universe, ratedItems, 4, 50, 0.5, 0.5);
        }

        return selected;
    }

    private List<Item> scoreFilterAndSortItems(RealVector locationVector,
                                               List<Item> universe,
                                               String breadth) {

        List<Item> filteredItems = new ArrayList<>();
        for (Item item : universe) {
            double cosine = locationVector.cosine(item.getVector());
            double dotProduct = locationVector.dotProduct(item.getVector());

            if (cosine > 0.0) {
                // Score items by their quality (vector magnitude) and similarity (cosine angle)
                // with the current location vector. ``breadth`` reflects how much we weight each
                // of these two factors. Wider breadths result in top-N lists with higher quality
                // items which are more diverse (less similar to the current location vector). This
                // results in better navigational items, but worse recommended items. Narrower
                // breadths result in top-N lists with lower (but still good) quality items which
                // are more similar to the current location vector (less diverse). This results in
                // worse navigational items, but better recommended items.
                if (breadth.equals("wide")) {
                    item.setScore(dotProduct * Math.pow(Math.abs(cosine), 1));
                } else if (breadth.equals("medium")) {
                    item.setScore(dotProduct * Math.pow(Math.abs(cosine), 2));
                } else if (breadth.equals("narrow")) {
                    item.setScore(dotProduct * Math.pow(Math.abs(cosine), 3));
                }

                filteredItems.add(item);
            }
        }
        filteredItems.sort(Comparator.comparing((Item i) -> i.getScore()).reversed());
        return filteredItems;
    }

    private void selectDiverseItems(List<Item> selected,
                                    List<Item> universe,
                                    List<Item> ratedItems,
                                    Integer n,
                                    Integer limit,
                                    Double dropout,
                                    Double ratedDropout) {

        int stopSize = selected.size() + n;
        while (selected.size() < stopSize) {
            List<Item> candidates = universe.stream()
                    // Exclude items we've already selected
                    .filter(x -> !selected.contains(x))
                    // Randomly exclude items to promote freshness
                    .filter(x -> random.nextDouble() >= dropout)
                    // Randomly exclude rated items to promote novel item discovery
                    .filter(x -> !ratedItems.contains(x) || random.nextDouble() >= ratedDropout)
                    // Only select from the highest-scoring, most relevant items
                    .limit(limit)
                    .collect(Collectors.toList());
```

```
211:
212:             // Select the candidate with the largest minimum distance to maximize diversity
213:             candidates.sort(Comparator.comparing((Item i) -> minDistance(i, selected)).reversed());
214:             selected.add(candidates.get(0));
215:         }
216:     }
217:
218:     private double minDistance(Item item, Collection<Item> otherItems) {
219:         // Get the minimum distance between this item and any of the other items
220:         return otherItems.stream()
221:                 .map(i -> i.getVector())
222:                 .mapToDouble(v -> v.getL1Distance(item.getVector()))
223:                 .min()
224:                 .orElse(-random.nextDouble());
225:     }
226: }
```