TopCoder Soybean Oracle Series - Soybean Marathon Match 1
Wladimir Leite (**wleite**)
01.06.2012 - 01.20.2012

Solution Description

Overview:

My solution is very simple. It compares each variant with all checks from the same experiment-location-repetition. Then it sums yield differences (between the variant and each check) and maturity time differences (those are added with negative weight, because less is better). I added few additional tweaks (bonus for specific years and for the type RR1, for example). Also added different weights to yield and maturity for each maturity band (extracted from the Location ID).

All the parameters were adjusted manually, by testing with random test cases created using the provided training set and analyzing the achieved score and its standard deviation.

Classify implementation:

My implementation to the `classify` method:

```
    public int[] classify(String[] expData, String[] locData) {
        Set<Experiment> experiments = getExperiments(expData); ①
        getLocations(locData); ②
        for (Experiment experiment : experiments) {
            analyseExperiment(experiment); ③
        }
        List<Variety> orderedVarieties = sortVarieties(experiments); ④
        return mountVarietiesIdArray(orderedVarieties); ⑤
    }
```

① Parses input experiment data. First it converts each `String` in the input array into a `DataRecord` object. After that, it reorganizes those `DataRecord` objects, creating a `Set` of `Experiment` objects, each one of them contains a `Set` of `Variety` objects. Each Variety contains a `Set` of `Trial` objects. The definition for those data classes are:

```
class Experiment {
    private final int id;
    private final int year;
    private final Set<Variety> varieties = new HashSet<Variety>();
}

class Variety implements Comparable<Variety> {
    private final int id;
    private final boolean check;
    private final boolean elite;
    private final VarietyType type;
    private final double relativeMaturity;
    private final Set<Trial> trials = new HashSet<Trial>();
    private double eliteScore;
}
class Trial {
```

```
    private final int loccd;
    private final int repetition;
    private final double yield;
    private final double maturiryNumber;
}
```

② Parses input location data into `Location` objects. There was a simplification here: it was completely disregarded the fact that a location could contain more than one entry on the input array. Because only the correspondence between Location ID and Maturity Band was used in the later steps of the solution, that seemed acceptable.

```
class Location {
    private final int loccd;
    private final int maturityBand;
    private final int maturitySubband;
    private final int maturityZone;

}
```

③ The `analyseExperiment` method is the main part of the solution, which gives a score for each variety based on its characteristics and the comparison with the check varieties used in the same experiment. It will be described in detail later.

④ Sorts the varieties of all experiments based on their "elite score". Before the sorting itself I added additional step that sorts varieties within an experiment and then adds an increasing penalty to the elite score of each variety based on its ranking inside the experiment (0 for the first variety, -30 for the second, -60 for the third and so on). The figure below illustrates this idea:

Score	Original Score		Score After Ranking Penalty	
	Exp 1	Exp 2	Exp 1	Exp 2
180	1		1	
170	2			
160				
150		5		5
140			2	
130				
120	3			
110		6		
100	4	7		
90				
80				6
70				
60			3	
50				
40				7
30				
20				
10			4	

⑤ Just mounts an array of varieties IDs from the already ordered list of `Variety` objects.

AnalyseExperiment implementation:

This method is responsible for calculating the “elite score” for each Variety. Higher score means that is more likely a variety is “elite”. As mentioned earlier, the implementation of this method and the weights used on it initially came from intuition, then they were validated and adjusted based on the results of a set of test cases with provided training data.

```
private void analyseExperiment(Experiment experiment) {
    Set<Variety> varieties = experiment.getVarieties();
    for (Variety testVariety : varieties) {
        if (testVariety.isCheck()) continue;
        double val = 0;
        Set<Trial> testTrials = testVariety.getTrials();
        List<Double> difYield = new ArrayList<Double>();
        List<Double> difMaturity = new ArrayList<Double>();
        for (Variety checkVariety : varieties) {
            if (!checkVariety.isCheck()) continue;
            Set<Trial> checkTrials = checkVariety.getTrials();
            for (Trial testTrial : testTrials) {
                for (Trial checkTrial : checkTrials) {
                    if (testTrial.getLoccd() != checkTrial.getLoccd()) continue;
                    if (testTrial.getRepetition() != checkTrial.getRepetition())  ①
                          continue;
                    if (testTrial.getYield() > 0 && checkTrial.getYield() > 0) {
                          difYield.add(getYieldWeight(testTrial.getLoccd()) *
                                  (testTrial.getYield() - checkTrial.getYield())); ②
                    }
                    if (testTrial.getMaturiryNumber() > 0 &&
                        checkTrial.getMaturiryNumber() > 0) {
                         difMaturity.add(getMaturityWeight(testTrial.getLoccd())
                                        * (testTrial.getMaturiryNumber() -
                                           checkTrial.getMaturiryNumber())); ③
                    }
                }
            }
        }
        Collections.sort(difYield);
        for (int i = 0; i < difYield.size() - 1; i++) {
            val += difYield.get(i); ④
        }
        if (difYield.size() > 0) val += difYield.get(0) * 4; ⑤

        Collections.sort(difMaturity);
        for (int i = difMaturity.size() - 1; i >= 0; i--) {
            val += difMaturity.get(i); ⑥
        }

        if (testVariety.getType().equals(VarietyType.RR1)) val += 100; ⑦
        if (difMaturity.size() <= 2) val -= 200; ⑧
        else if (difMaturity.size() <= 4) val -= 150;
        else if (difMaturity.size() <= 6) val -= 100;
        else if (difMaturity.size() <= 8) val -= 10;

        if (experiment.getYear() == 2002) val *= 4; ⑨
        if (experiment.getYear() == 2003) val *= 3;
        testVariety.setEliteScore(val);
    }
}
```

① Loops through all varieties of the specified `Experiment`, comparing each `Trial` of the `Variety` with each `Trial` of the “checks” from the same experiment, with the same Location ID and repetition.

② Computes the difference between the variety yield and the check yield, multiplied by a weight that depends on maturity band (extracted from the Location ID), storing in on a list. Initially this weight was fixed as 1.2. After I tried different weights for each band (from 0.5 to 1.4) and took the best for each band:

```
0=0.6, 1=1.4, 2=1.3, 3=1.3, 4=1.3, 5=1.4, 6=0.8, 7=0.5, 8=1.2, 9=1.0, 10=1.0
```

③ The same as above from the maturity number. In this case initial weight is -0.8 (negative, because the lower the better). After testing the final weights, for each band, are:

```
0=-1.4,1=-1.3,2=-1.3,3=-0.8,4=-0.8,5=-0.5,6=-0.5,7=-1.3,8=-0.7,9=-0.8,10=-0.8
```

④ Sums all differences calculated on step 2 (yield), ignoring the largest one. Ignoring the largest value came from results of tests only, because it seems counterintuitive.

⑤ Adds with a weight of 4 the worse (lower) value obtained on step 2.

⑥ Sums all differences calculated on step 3 (maturity number).

⑦ Adds a bonus of 100 points if the variety type is RR1.

⑧ Adds a penalty for varieties with few results from step 3. Those varieties were not tested enough and are likely to not be chosen as elite.

⑨ Adds a bonus for varieties from years 2002 and 2003. The weights applied here (multiplication by 4 and 3 respectively) seemed too high and maybe over fitted, but since they affect a lot the score in local tests (around 25% loss if removed), I decided to keep them.