Weka[25] Bagging 源代码分析

作者：Koala++/屈伟

先翻译一段 Bagging 的介绍，Breiman 的 bagging 算法，是 bootstrap aggregating 的缩写，是最早的 Ensemble 算法之一，它也是最直接容易实现，又有着另人惊讶的好效果的算法之一。Bagging 中的多样性是由有放回抽取训练样本来实现的，用这种方式随机产生多个训练数据的子集，在每一个训练集的子集上训练一个同种分类器，最终分类结果是由多个分类器的分类结果多数投票而产生的。Breiman's bagging, short for bootstrap aggregating, is one of the earliest ensemble based algorithms. It is also one of the most intuitive and simplest to implement, with a surprisingly good performance . Diversity in bagging is obtained by using bootstrapped replicas of the training data: different training data subsets are randomly drawn—with replacement—from the entire training data. Each training data subset is used to train a different classifier of the same type. Individual classifiers are then combined by taking a majority vote of their decisions. For any given instance, the class chosen by most classifiers is the ensemble decision.

Bagging 类在 weka.classifiers.meta 包下面。Bagging 继承自 RandomizeableInteratedSingleClassifierEnhancer，而它又继承自 IteratedSingleClassifierEnhancer，它再继承自 SingleClassifierEnhancer，最后一个继承自 Classifier。我的 UML 工具似乎过期了，有空补上。

看一下构造函数：

```
public Bagging() {
    m_Classifier = new weka.classifiers.trees.REPTree();
}
```

可以看到默认的基分类器是 REPTree。

接下来看 buildClassifier 函数：

```
// can classifier handle the data?
getCapabilities().testWithFail(data);

// remove instances with missing class
data = new Instances(data);
data.deleteWithMissingClass();

super.buildClassifier(data);

if (m_CalcOutOfBag && (m_BagSizePercent != 100)) {
    throw new IllegalArgumentException("Bag size needs to be 100% if "
           + "out-of-bag error is to be calculated!");
}
```

只有一行代码值得看一下 super.buildClassifier：

```
public void buildClassifier(Instances data) throws Exception {

    if (m_Classifier == null) {
        throw new Exception("A base classifier has not been specified!");
    }
    m_Classifiers = Classifier.makeCopies(m_Classifier,
        m_NumIterations);
}
```

这里将 m_Classifier 复制 m_NumIterations 份到 m_Classifiers 数组中去。

```
int bagSize = data.numInstances() * m_BagSizePercent / 100;
Random random = new Random(m_Seed);

boolean[][] inBag = null;
if (m_CalcOutOfBag)
    inBag = new boolean[m_Classifiers.length][];
```

bagSize 是一个 Bag 的大小，也就是它里面有多少样本。

```
for (int j = 0; j < m_Classifiers.length; j++) {
    Instances bagData = null;

    // create the in-bag dataset
    if (m_CalcOutOfBag) {
        inBag[j] = new boolean[data.numInstances()];
        bagData = resampleWithWeights(data, random, inBag[j]);
    } else {
        bagData = data.resampleWithWeights(random);
        if (bagSize < data.numInstances()) {
            bagData.randomize(random);
            Instances newBagData = new Instances(bagData, 0, bagSize);
            bagData = newBagData;
        }
    }

    if (m_Classifier instanceof Randomizable) {
        ((Randomizable) m_Classifiers[j]).setSeed(random.nextInt());
    }

    // build the classifier
    m_Classifiers[j].buildClassifier(bagData);
}
```

暂时不去看m_CalcOutOfBag的情况，当然最关键的是resampleWithWeights：

```
/**
 * Creates a new dataset of the same size using random sampling with
 * replacement according to the current instance weights. The weights of
 * the instances in the new dataset are set to one.
*/
public Instances resampleWithWeights(Random random) {

    double[] weights = new double[numInstances()];
    for (int i = 0; i < weights.length; i++) {
        weights[i] = instance(i).weight();
    }
    return resampleWithWeights(random, weights);
}
```

注释上写的是根据当前样本的权重用有放回取样的方法创建一个同样大小的新数据集，新数据集中的样本权重为 1。这里先是把权重记录下来，再用一个重载函数去做：

接下来是看数据集中的样本是否大于 bagSize，如果不大于，其实就没什么意思了。如果大于，再把 bagData 随机一次，取前面的 bagSize 个样本，下面如果 m_Classifier 是 Randomizable 的一个实例，那么就给它再指定一个新的随机种子，这点很关键，自己写的时候，常常忘记。最后训练第 j 个分类器。

现在再看 resampleWithWeights：

```
public Instances resampleWithWeights(Random random, double[] weights) {
```

```
    if (weights.length != numInstances()) {
        throw new IllegalArgumentException(
               "weights.length != numInstances.");
    }
    Instances newData = new Instances(this, numInstances());
    if (numInstances() == 0) {
        return newData;
    }
    double[] probabilities = new double[numInstances()];
    double sumProbs = 0, sumOfWeights = Utils.sum(weights);
    for (int i = 0; i < numInstances(); i++) {
        sumProbs += random.nextDouble();
        probabilities[i] = sumProbs;
    }
    Utils.normalize(probabilities, sumProbs / sumOfWeights);

    // Make sure that rounding errors don't mess things up
    probabilities[numInstances() - 1] = sumOfWeights;
    int k = 0;
    int l = 0;
    sumProbs = 0;
    while ((k < numInstances() && (l < numInstances()))) {
        if (weights[l] < 0) {
            throw new IllegalArgumentException(
                   "Weights have to be positive.");
        }
        sumProbs += weights[l];
        while ((k < numInstances()) && (probabilities[k] <= sumProbs)) {
            newData.add(instance(l));
            newData.instance(k).setWeight(1);
            k++;
        }
        l++;
    }
    return newData;
}
```

sumProbs 是产生的随机数的总和，而 probabilities 是第 i 次的总和，Utils.normalize 的代码如下：

```
public static void normalize(double[] doubles, double sum) {
    if (Double.isNaN(sum)) {
        throw new IllegalArgumentException(
               "Can't normalize array. Sum is NaN.");
    }
    if (sum == 0) {
        // Maybe this should just be a return.
        throw new IllegalArgumentException(
               "Can't normalize array. Sum is zero.");
    }
    for (int i = 0; i < doubles.length; i++) {
        doubles[i] /= sum;
    }
}
```

这一步是将所产生的随机数与权重对应起来，因为产生的probabilities在(0,1)范围内，可能与样本权重对应不起来，在下面的二重循环中，看到sumProbs重新记数，它的意义就是加上weights[l]之后，probability[k] 如果到不到相应的sumProbs，就重复地加这一个相同的样本。

通过这种方式来产生有放回的取样样本。

现在看m_CalcOutOfBag为true的时候，首先会有一个inBag二维数组，第一维大小为分类器个数，第二维为样本个数。public final Instances resampleWithWeights(Instances data, Random random, boolean[] sampled)这个函数与Intances中的差不多，只多了一句话就是sampled[I] = true，表示这个样本采样时有它。接下来看buildClassifier的后面一部分，看起来很长，其实蛮简单的。

```
// calc OOB error?
if (getCalcOutOfBag()) {
    double outOfBagCount = 0.0;
    double errorSum = 0.0;
    boolean numeric = data.classAttribute().isNumeric();

    for (int i = 0; i < data.numInstances(); i++) {
        double vote;
        double[] votes;
        if (numeric)
            votes = new double[1];
        else
            votes = new double[data.numClasses()];

        // determine predictions for instance
        int voteCount = 0;
        for (int j = 0; j < m Classifiers.length; j++) {
            if (inBag[j][i])
                continue;

            voteCount++;
            double pred = m Classifiers[j].classifyInstance(data
                    .instance(i));
            if (numeric)
                votes[0] += pred;
            else
                votes[(int) pred]++;
        }

        // "vote"
        if (numeric) {
            vote = votes[0];
            if (voteCount > 0) {
                vote /= voteCount; // average
            }
        } else {
            vote = Utils.maxIndex(votes); // majority vote
        }

        // error for instance
        outOfBagCount += data.instance(i).weight();
        if (numeric) {
            errorSum += StrictMath.abs(vote
                    - data.instance(i).classValue())
                    * data.instance(i).weight();
        } else {
            if (vote != data.instance(i).classValue())
                errorSum += data.instance(i).weight();
        }
    }
```

```
    m_OutOfBagError = errorSum / outOfBagCount;
} else {
    m_OutOfBagError = 0;
}
```

这里 inBag 就可以判断哪几个分类器学习的时候有某一个样本，看 out of bag 错误率的时候，也就是用那些在学习时没有见过这个样本的分类器去分类这个样本，再用多数投票(majority vote)的方法决定分类结果。对于数值型的属性，就是将结果减去真实值，再乘权重，而对于离散型属性，只需要在分类错时，乘以权重累加到 errorSum 上。