Weka[22] REPTree 源代码分析

作者：Koala++/屈伟

如果你分析完了ID3，还想进一步学习，最好还是先学习REPTree，它没有牵扯到那么多类，两个类完成了全部的工作，看起来比较清楚，J48虽然有很强的可扩展性，但是初看起来还是有些费力，REPTree也是我卖算法时(为了买一台运算能力强一点的计算机，我也不得不赚钱)，顺便分析的，但因为我以前介绍过J48了，重复的东西不想再次介绍了，如果有什么不明白的，就把我两篇写的结合起来看吧。

我们再次从buildClassifier开始。

```
Random random = new Random(m_Seed);

m_zeroR = null;
if (data.numAttributes() == 1) {
    m_zeroR = new ZeroR();
    m_zeroR.buildClassifier(data);
    return;
}
```

如果就只有一个属性，也就是类别属性，就用 ZeroR 分类器学习，ZeroR 分类器返回训练集中出现最多的类别值，已经讲过了 Weka 开发[15]。

```
// Randomize and stratify
data.randomize(random);
if (data.classAttribute().isNominal()) {
    data.stratify(m_NumFolds);
}
```

randomize 就是把 data 中的数据重排一下，如果类别属性是离散值，那么用 stratify 函数，stratify 意思是分层，现在把这个函数列出来：

```
public void stratify(int numFolds) {
    if (classAttribute().isNominal()) {
        // sort by class
        int index = 1;
        while (index < numInstances()) {
            Instance instance1 = instance(index - 1);
            for (int j = index; j < numInstances(); j++) {
                Instance instance2 = instance(j);
                if ((instance1.classValue() == instance2.classValue())
                        || (instance1.classIsMissing() && instance2
                                .classIsMissing())) {
                    swap(index, j);
                    index++;
                }
            }
            index++;
        }
        stratStep(numFolds);
    }
}
```

上面这两重循环，就是根据类别值进行冒泡。下面有调用了 stratStep 函数：

```
protected void stratStep(int numFolds) {

    FastVector newVec = new FastVector(m_Instances.capacity());
    int start = 0, j;
```

```
    // create stratified batch
    while (newVec.size() < numInstances()) {
        j = start;
        while (j < numInstances()) {
            newVec.addElement(instance(j));
            j = j + numFolds;
        }
        start++;
    }
    m_Instances = newVec;
}
```

这里我举一个例子说明：j=0 时，numFolds 为 10 时，newVec 加入的 instance 下标就为 0,10,20…。这样的好处就是我们把各种类别的样本类似平均分布了。

```
// Split data into training and pruning set
Instances train = null;
Instances prune = null;
if (!m_NoPruning) {
    train = data.trainCV(m_NumFolds, 0, random);
    prune = data.testCV(m_NumFolds, 0);
} else {
    train = data;
}
```

关于 trainCV 这个就不讲了，就是 crossValidation 的第 0 个训练集作为这次的训练集(train)。而作为剪枝的数据集 prune 为第 0 个测试集。

```
// Create array of sorted indices and weights
int[][] sortedIndices = new int[train.numAttributes()][0];
double[][] weights = new double[train.numAttributes()][0];
double[] vals = new double[train.numInstances()];
for (int j = 0; j < train.numAttributes(); j++) {
    if (j != train.classIndex()) {
        weights[j] = new double[train.numInstances()];
        if (train.attribute(j).isNominal()) {

            // Handling nominal attributes. Putting indices of
            // instances with missing values at the end.
            sortedIndices[j] = new int[train.numInstances()];
            int count = 0;
            for (int i = 0; i < train.numInstances(); i++) {
                Instance inst = train.instance(i);
                if (!inst.isMissing(j)) {
                    sortedIndices[j][count] = i;
                    weights[j][count] = inst.weight();
                    count++;
                }
            }
            for (int i = 0; i < train.numInstances(); i++) {
                Instance inst = train.instance(i);
                if (inst.isMissing(j)) {
                    sortedIndices[j][count] = i;
                    weights[j][count] = inst.weight();
                    count++;
                }
            }
        } else {
            // Sorted indices are computed for numeric attributes
```

```
            for (int i = 0; i < train.numInstances(); i++) {
                Instance inst = train.instance(i);
                vals[i] = inst.value(j);
            }
            sortedIndices[j] = Utils.sort(vals);
            for (int i = 0; i < train.numInstances(); i++) {
                weights[j][i] = train.instance(sortedIndices[j][i])
                        .weight();
            }
        }
    }
}
```

sortedIndices 表示第 j 属性的第 count 个样本下标是多少，weights 表示第 j 个属性第 count 个样本的权重，如果 j 属性是离散值，通过两个 for 循环，在 sortedIndices 和 weights 中在 j 属性上是缺失值的样本就排在了后面。如果是连续值，那么就把全部样本 j 属性值得到，再排序，最后记录权重。

```
// Compute initial class counts
double[] classProbs = new double[train.numClasses()];
double totalWeight = 0, totalSumSquared = 0;
for (int i = 0; i < train.numInstances(); i++) {
    Instance inst = train.instance(i);
    if (data.classAttribute().isNominal()) {
        classProbs[(int) inst.classValue()] += inst.weight();
        totalWeight += inst.weight();
    } else {
        classProbs[0] += inst.classValue() * inst.weight();
        totalSumSquared += inst.classValue() * inst.classValue()
                * inst.weight();
        totalWeight += inst.weight();
    }
}
m_Tree = new Tree();
double trainVariance = 0;
if (data.classAttribute().isNumeric()) {
    trainVariance = m_Tree.singleVariance(classProbs[0],
            totalSumSquared, totalWeight) / totalWeight;
    classProbs[0] /= totalWeight;
}
```

计算初始化类别概率，如果类别是离散值，classProbs 中记录的是属性类别 inst.classValue()的样本权重之和，totalWeight 是全部样本权重和。如果类别是连续值，classProbs[0]中是权重乘以类别值，它还有一个 totalSumSquared 是类别值平方乘以权重之和。

m_Tree 是一个 Tree 对象，如果是连续值类别，用 m_Tree 的成员函数来计算 trainVariance 这个带权重的方差，最后 classProbs[0]相当于期望。

```
// Build tree
m_Tree.buildTree(sortedIndices, weights, train, totalWeight,
        classProbs, new Instances(train, 0), m_MinNum,
        m_MinVariancePropㅤ* trainVariance, 0, m_MaxDepth);
```

好了，终于可以建树了，除了 VC，我还真没怎么见过这么多参数。现在把它拆开分析：

```
// Store structure of dataset, set minimum number of instances
// and make space for potential info from pruning data
m_Info = header;
m_HoldOutDist = new double[data.numClasses()];

// Make leaf if there are no training instances
int helpIndex = 0;
```

```
if (data.classIndex() == 0) {
    helpIndex = 1;
}
if (sortedIndices[helpIndex].length == 0) {
    if (data.classAttribute().isNumeric()) {
        m Distribution = new double[2];
    } else {
        m Distribution = new double[data.numClasses()];
    }
    m ClassProbs = null;
    return;
}
```

m_Info 保存的是数据集的表头结构，m_HoldOutDist 后面会讲到，是用于剪枝的。这面这个有点意思，helpIndex 在类别 index 不是 0 的情况下是 1，否则是 0，因为 sortedIndices 中没有类别列。初始化 m_Distribution，如果是连续值，数组长度是 2，第一个保存方差，后面是样本总权重。离散值不会说，当然是类别值个数。

```
double priorVar = 0;
if (data.classAttribute().isNumeric()) {

    // Compute prior variance
    double totalSum = 0, totalSumSquared = 0, totalSumOfWeights = 0;
    for (int i = 0; i < sortedIndices[helpIndex].length; i++) {
        Instance inst = data.instance(sortedIndices[helpIndex][i]);
        totalSum += inst.classValue() * weights[helpIndex][i];
        totalSumSquared += inst.classValue() * inst.classValue()
                * weights[helpIndex][i];
        totalSumOfWeights += weights[helpIndex][i];
    }
    priorVar = singleVariance(totalSum, totalSumSquared,
            totalSumOfWeights);
}
```

这个就非常简单了，如果类别是连续值。再说一下，这里 helpIndex 无所谓，只要不是类别 index 就好。totalSum 是类别值与样本权重的乘积和，totalSumSquared 是类别值平方乘样本权重和，totalSumOfWeights 是权重和。这里还是说一下，singleVariance 就是变换后的期望计算公式。

```
// Check if node doesn't contain enough instances, is pure
// or the maximum tree depth is reached
m ClassProbs = new double[classProbs.length];
System.arraycopy(classProbs, 0, m ClassProbs, 0, classProbs.length);
if ((totalWeight < (2 * minNum))
        ||

        // Nominal case
        (data.classAttribute().isNominal() && Utils.eq(
                m ClassProbs[Utils.maxIndex(m ClassProbs)], Utils
                        .sum(m ClassProbs)))
        ||

        // Numeric case
        (data.classAttribute().isNumeric() && ((priorVar / totalWeight)
             < minVariance))
        ||

        // Check tree depth
        ((m_MaxDepth >= 0) && (depth >= maxDepth))) {
```

```
    // Make leaf
    m_Attribute = -1;
    if (data.classAttribute().isNominal()) {

        // Nominal case
        m_Distribution = new double[m_ClassProbs.length];
        for (int i = 0; i < m_ClassProbs.length; i++) {
            m_Distribution[i] = m_ClassProbs[i];
        }
        Utils.normalize(m_ClassProbs);
    } else {

        // Numeric case
        m_Distribution = new double[2];
        m_Distribution[0] = priorVar;
        m_Distribution[1] = totalWeight;
    }
    return;
}
```

先看一下不会再分裂的情况，第一种，总样本权重还不到最小分裂样本数的 2 倍(因为至少要分出来两个子结点嘛)，第二种，类别是离散值的情况下，如果样本都属于一个类别(以前讲过为什么)。第三种，类别是连续值的情况下，如果方差小于一个最小方差，最小方差是由一个定义的常数与总方差的积。最后一种如果超过了定义的树的深度。

如果是离散值，就将 m_ClassProbs 数组中的内容复制到 m_Distribution 中，再进行规范化，如果是连续值，把方差和总权重保存。

```
// Compute class distributions and value of splitting
// criterion for each attribute
double[] vals = new double[data.numAttributes()];
double[][][] dists = new double[data.numAttributes()][0][0];
double[][] props = new double[data.numAttributes()][0];
double[][] totalSubsetWeights = new double[data.numAttributes()][0];
double[] splits = new double[data.numAttributes()];
if (data.classAttribute().isNominal()) {

    // Nominal case
    for (int i = 0; i < data.numAttributes(); i++) {
        if (i != data.classIndex()) {
            splits[i] = distribution(props, dists, i,
                    sortedIndices[i], weights[i],
                    totalSubsetWeights, data);
            vals[i] = gain(dists[i], priorVal(dists[i]));
        }
    }
} else {

    // Numeric case
    for (int i = 0; i < data.numAttributes(); i++) {
        if (i != data.classIndex()) {
            splits[i] = numericDistribution(props, dists, i,
                    sortedIndices[i], weights[i],
                    totalSubsetWeights, data, vals);
        }
    }
}
```

这里出现了一下 ditribution 函数，也是非常长，但是又很重要，所以我还是先介绍它：

```
double splitPoint = Double.NaN;
Attribute attribute = data.attribute(att);
double[][] dist = null;
int i;

if (attribute.isNominal()) {

    // For nominal attributes
    dist = new double[attribute.numValues()][data.numClasses()];
    for (i = 0; i < sortedIndices.length; i++) {
        Instance inst = data.instance(sortedIndices[i]);
        if (inst.isMissing(att)) {
            break;
        }
        dist[(int) inst.value(att)][(int) inst.classValue()] +=
            weights[i];
    }
}
```

先讲一下离散值的情况，实现与 j48 包下面的 Distribution 非常相似，dist 第一维是属性值，第二维是类别值，元素值是样本权重累加值。

```
else {
    // For numeric attributes
    double[][] currDist = new double[2][data.numClasses()];
    dist = new double[2][data.numClasses()];

    // Move all instances into second subset
    for (int j = 0; j < sortedIndices.length; j++) {
        Instance inst = data.instance(sortedIndices[j]);
        if (inst.isMissing(att)) {
            break;
        }
        currDist[1][(int) inst.classValue()] += weights[j];
    }
    double priorVal = priorVal(currDist);
    System.arraycopy(currDist[1], 0, dist[1], 0, dist[1].length);

    // Try all possible split points
    double currSplit = data.instance(sortedIndices[0]).value(att);
    double currVal, bestVal = -Double.MAX_VALUE;
    for (i = 0; i < sortedIndices.length; i++) {
        Instance inst = data.instance(sortedIndices[i]);
        if (inst.isMissing(att)) {
            break;
        }
        if (inst.value(att) > currSplit) {
            currVal = gain(currDist, priorVal);
            if (currVal > bestVal) {
                bestVal = currVal;
                splitPoint = (inst.value(att) + currSplit) / 2.0;
                for (int j = 0; j < currDist.length; j++) {
                    System.arraycopy(currDist[j], 0, dist[j], 0,
                           dist[j].length);
                }
            }
        }
        currSplit = inst.value(att);
        currDist[0][(int) inst.classValue()] += weights[i];
```

```
            currDist[1][(int) inst.classValue()] -= weights[i];
        }
    }
```

不想讲了，和 J48 也是一样，先把样本存在后一子结点中 currDist[1]，然后依次试属性值，找到一个最好看分裂点。

```
// Compute weights
props[att] = new double[dist.length];
for (int k = 0; k < props[att].length; k++) {
    props[att][k] = Utils.sum(dist[k]);
}
if (!(Utils.sum(props[att]) > 0)) {
    for (int k = 0; k < props[att].length; k++) {
        props[att][k] = 1.0 / (double) props[att].length;
    }
} else {
    Utils.normalize(props[att]);
}
```

props 中保存的就是第 att 个属性的第 k 个属性值的样本权重之和。如果这个值不太于 0，就给它赋值为 1 除以这个属性的全部可能取值。否则规范化。

```
// Distribute counts
while (i < sortedIndices.length) {
    Instance inst = data.instance(sortedIndices[i]);
    for (int j = 0; j < dist.length; j++) {
        dist[j][(int) inst.classValue()] += props[att][j]
                * weights[i];
    }
    i++;
}

// Compute subset weights
subsetWeights[att] = new double[dist.length];
for (int j = 0; j < dist.length; j++) {
    subsetWeights[att][j] += Utils.sum(dist[j]);
}

// Return distribution and split point
dists[att] = dist;
return splitPoint;
```

i 这里初始是有确定属性值与缺失值的分界下标值，开始一时头晕还没看出来，调试才看出来。如果有缺失值，就用每一个属性值都加上相应的权重来代替。在 att 属性上分裂，那种子结点的权重和为 dist 在 j 这种属性取值上的和。最后把 dist 赋值给 dists[att]，返回分裂点。

现在再跳回到 buildTree 函数，接着讲 gain 函数就是计算信息增益，不讲了。numericDistribution 还是这么长，而且也差不多，也就算了吧。

```
// Find best attribute
m Attribute = Utils.maxIndex(vals);
int numAttVals = dists[m Attribute].length;

// Check if there are at least two subsets with
// required minimum number of instances
int count = 0;
for (int i = 0; i < numAttVals; i++) {
    if (totalSubsetWeights[m Attribute][i] >= minNum) {
        count++;
    }
```

```
    if (count > 1) {
        break;
    }
}
```

vals 中信息增益值，m_Attribute 就是有最大信息增益值的属性下标，再下来看是否这个属性可以分出两个大于 minNum 样本数的子结点。

```
// Any useful split found?
if ((vals[m_Attribute] > 0) && (count > 1)) {

    // Build subtrees
    m_SplitPoint = splits[m_Attribute];
    m_Prop = props[m_Attribute];
    int[][][] subsetIndices = new int[numAttVals][data
           .numAttributes()][0];
    double[][][] subsetWeights = new double[numAttVals][data
           .numAttributes()][0];
    splitData(subsetIndices, subsetWeights, m_Attribute,
           m_SplitPoint, sortedIndices, weights, data);
    m_Successors = new Tree[numAttVals];
    for (int i = 0; i < numAttVals; i++) {
        m_Successors[i] = new Tree();
        m_Successors[i].buildTree(subsetIndices[i],
                subsetWeights[i], data,
                totalSubsetWeights[m_Attribute][i],
                dists[m_Attribute][i], header, minNum, minVariance,
                depth + 1, maxDepth);
    }
} else {

    // Make leaf
    m_Attribute = -1;
}
```

如果找到了可以分裂的属性，那我们就可以建立了树了，看起来乱七八糟很复杂的样子，其实如果你把上面讲的搞清楚了，这里和 ID3，J48 没有什么区别。如果不能分裂，就把 m_Attribute 置 1，标记一下。

```
// Normalize class counts
if (data.classAttribute().isNominal()) {
    m_Distribution = new double[m_ClassProbs.length];
    for (int i = 0; i < m_ClassProbs.length; i++) {
        m_Distribution[i] = m_ClassProbs[i];
    }
    Utils.normalize(m_ClassProbs);
} else {
    m_Distribution = new double[2];
    m_Distribution[0] = priorVar;
    m_Distribution[1] = totalWeight;
}
```

这个其实没什么好讲的，只是赋值到 m_Distribution，建树就已经讲完了。但是在 buildClassifier 我们还剩下三行，是关于剪枝的，当时在介绍 J48 的时候，就没有讲，因为我不需要用那部分，当时也没怎么看。

```
// Insert pruning data and perform reduced error pruning
if (!m_NoPruning) {
    m_Tree.insertHoldOutSet(prune);
    m_Tree.reducedErrorPrune();
    m_Tree.backfitHoldOutSet(prune);
```

```
}
```

如果非不剪枝，那么就是剪枝了，先看第一个被调用的函数：

```
protected void insertHoldOutSet(Instances data) throws Exception {

    for (int i = 0; i < data.numInstances(); i++) {
        insertHoldOutInstance(data.instance(i), data.instance(i)
                .weight(), this);
    }
}
```

prune 数据集中的每一个样本作为参数调用 insertHoldOutInstance，它也有点长，把它一部分一部分列出来：

```
// Insert instance into hold-out class distribution
if (inst.classAttribute().isNominal()) {

    // Nominal case
    m HoldOutDist[(int) inst.classValue()] += weight;
    int predictedClass = 0;
    if (m ClassProbs == null) {
        predictedClass = Utils.maxIndex(parent.m ClassProbs);
    } else {
        predictedClass = Utils.maxIndex(m ClassProbs);
    }
    if (predictedClass != (int) inst.classValue()) {
        m HoldOutError += weight;
    }
} else {

    // Numeric case
    m HoldOutDist[0] += weight;
    double diff = 0;
    if (m ClassProbs == null) {
        diff = parent.m ClassProbs[0] - inst.classValue();
    } else {
        diff = m ClassProbs[0] - inst.classValue();
    }
    m HoldOutError += diff * diff * weight;
}
```

看一下离散的情况，如果是离散类别，看它预测出的类别是否与真实类别相同，如果不同，就把样本权重累加到 m_HoldOutError 上，其中==null 的情况应该是这个叶子结点上曾经分的时候就没样本。在连续类别时，是把预测值与真实值的差的平方乘权重加到 m_holdOutError 上，

```
// The process is recursive
if (m Attribute != -1) {

    // If node is not a leaf
    if (inst.isMissing(m Attribute)) {

        // Distribute instance
        for (int i = 0; i < m Successors.length; i++) {
            if (m Prop[i] > 0) {
                m Successors[i].insertHoldOutInstance(inst, weight
                        * m Prop[i], this);
            }
        }
    } else {
```

```
        if (m_Info.attribute(m_Attribute).isNominal()) {

            // Treat nominal attributes
            m_Successors[(int) inst.value(m_Attribute)]
                    .insertHoldOutInstance(inst, weight, this);
        } else {

            // Treat numeric attributes
            if (inst.value(m_Attribute) < m_SplitPoint) {
                m_Successors[0].insertHoldOutInstance(inst, weight,
                        this);
            } else {
                m_Successors[1].insertHoldOutInstance(inst, weight,
                        this);
            }
        }
    }
}
```

m_Attribute 等于-1 时就是叶子结点，前面已经讲过了，如果是缺失值的情况，又是把所有可能算一遍(前两天看论文，有一篇论文提到对缺失值的运行，在 C4.5 中占到了 80%的时间)。如果不是缺失值就递归。这个函数整体的含义就是计算父结点和子结点，为最后看分还是不分好做准备。

好了，看第二个函数：

```
protected double reducedErrorPrune() throws Exception {

    // Is node leaf ?
    if (m_Attribute == -1) {
        return m_HoldOutError;
    }

    // Prune all sub trees
    double errorTree = 0;
    for (int i = 0; i < m_Successors.length; i++) {
        errorTree += m_Successors[i].reducedErrorPrune();
    }

    // Replace sub tree with leaf if error doesn't get worse
    if (errorTree >= m_HoldOutError) {
        m_Attribute = -1;
        m_Successors = null;
        return m_HoldOutError;
    } else {
        return errorTree;
    }
}
```

如果开始就是叶子结点，太不可思议了，直接返回。接下来，这是一个递归，递归就在做一件事情，如果几个子结点的错误加起来比父结点还高，意思也就是说分裂比不分裂还要差，那么我们就把子结点剪去，也就是剪枝，在这里是剪叶子？剪枝的时候，设置 m_Attribute，然后把子结点置空，返回父结点的错误值。

最后一个函数：

```
protected void backfitHoldOutSet(Instances data) throws Exception {

    for (int i = 0; i < data.numInstances(); i++) {
        backfitHoldOutInstance(data.instance(i), data.instance(i)
                .weight(), this);
```

```
    }
}
```

backfitHoldOutInstance 不难，但是还有有点长，分开贴出来：

```
// Insert instance into hold-out class distribution
if (inst.classAttribute().isNominal()) {

    // Nominal case
    if (m_ClassProbs == null) {
        m_ClassProbs = new double[inst.numClasses()];
    }
    System.arraycopy(m_Distribution, 0, m_ClassProbs, 0, inst
            .numClasses());
    m_ClassProbs[(int) inst.classValue()] += weight;
    Utils.normalize(m_ClassProbs);
} else {

    // Numeric case
    if (m_ClassProbs == null) {
        m_ClassProbs = new double[1];
    }
    m_ClassProbs[0] *= m_Distribution[1];
    m_ClassProbs[0] += weight * inst.classValue();
    m_ClassProbs[0] /= (m_Distribution[1] + weight);
}
```

这个函数主要是把以前用训练集测出来的值，现在把剪枝集的样本信息也加进去。这些以前也都讲过。

```
// The process is recursive
if (m_Attribute != -1) {

    // If node is not a leaf
    if (inst.isMissing(m_Attribute)) {

        // Distribute instance
        for (int i = 0; i < m_Successors.length; i++) {
            if (m_Prop[i] > 0) {
                m_Successors[i].backfitHoldOutInstance(inst, weight
                        * m_Prop[i], this);
            }
        }
    } else {

        if (m_Info.attribute(m_Attribute).isNominal()) {

            // Treat nominal attributes
            m_Successors[(int) inst.value(m_Attribute)]
                    .backfitHoldOutInstance(inst, weight, this);
        } else {

            // Treat numeric attributes
            if (inst.value(m_Attribute) < m_SplitPoint) {
                m_Successors[0].backfitHoldOutInstance(inst,
                        weight, this);
            } else {
                m_Successors[1].backfitHoldOutInstance(inst,
                        weight, this);
            }
        }
```

```
    }
}
```

不想讲了，自己看吧，distributionForInstance 也不讲了，如果是一直看我的东西过来的，到现在还不明白，我也没话说了。