weka[29] Logistic 源代码分析

作者：Koala++/屈伟

Logistic Regression 是非常重要的一个算法，可以从 Tom Mitchell 的主页上把 new chapter 的第一章看一下，或是 Ng Andrew 的 lecture notes 的 Part II 看一下。

从 buildClassifier 开始：

```
if (train.classAttribute().type() != Attribute.NOMINAL) {
    throw new UnsupportedClassTypeException(
            "Class attribute must be nominal.");
}
if (train.checkForStringAttributes()) {
    throw new UnsupportedAttributeTypeException(
            "Can't handle string attributes!");
}
train = new Instances(train);
train.deleteWithMissingClass();
if (train.numInstances() == 0) {
    throw new IllegalArgumentException(
            "No train instances without missing class value!");
}
```

类别属性必须是离散的，不能处理字符串属性，删除类别缺失的样本，删除后样本数不能为 0。

```
// Replace missing values
m_ReplaceMissingValues = new ReplaceMissingValues();
m_ReplaceMissingValues.setInputFormat(train);
train = Filter.useFilter(train, m_ReplaceMissingValues);

// Remove useless attributes
m_AttFilter = new RemoveUseless();
m_AttFilter.setInputFormat(train);
train = Filter.useFilter(train, m_AttFilter);

// Transform attributes
m_NominalToBinary = new NominalToBinary();
m_NominalToBinary.setInputFormat(train);
train = Filter.useFilter(train, m_NominalToBinary);
```

替换缺失值，删除无用的属性，转换成二值属性。

```
// Extract data
m_ClassIndex = train.classIndex();
m_NumClasses = train.numClasses();

int nK = m_NumClasses - 1; // Only K-1 class labels needed
int nR = m_NumPredictors = train.numAttributes() - 1;
int nC = train.numInstances();

m_Data = new double[nC][nR + 1]; // Data values
int[] Y = new int[nC]; // Class labels
double[] xMean = new double[nR + 1]; // Attribute means
double[] xSD = new double[nR + 1]; // Attribute stddev's
double[] sY = new double[nK + 1]; // Number of classes
double[] weights = new double[nC]; // Weights of instances
double totWeights = 0; // Total weights of the instances
```

```
m_Par = new double[nR + 1][nK]; // Optimized parameter values
```

看一下有哪些值，输入属性值，类标签，属性均值，属性标准差，类别数，样本权重，样本的总权重，优化后的参数值。

```
for (int i = 0; i < nC; i++) {
    // initialize X[][]
    Instance current = train.instance(i);
    Y[i] = (int) current.classValue(); // Class value starts from 0
    weights[i] = current.weight(); // Dealing with weights
    totWeights += weights[i];

    m_Data[i][0] = 1;
    int j = 1;
    for (int k = 0; k <= nR; k++) {
        if (k != m_ClassIndex) {
            double x = current.value(k);
            m_Data[i][j] = x;
            xMean[j] += weights[i] * x;
            xSD[j] += weights[i] * x * x;
            j++;
        }
    }

    // Class count
    sY[Y[i]]++;
}
```

nC 是样本数，Y[i]记录下每个样本的类别值，类别值从 0 开始，weight 记录下当前样本的权重，totWeights 统计数权重，m_Data 第二维是从 1 开始记录属性值的，第一个值是 1，也就是公式中 sum_0^n(theta(i)*x(i))，从 0 开始那么也就是 x0 为 0。xMean[j]现在累计第 j 个属性的属性值*权重，xSD 累计属性值平方*权重。sY 是统计 Y[i]属性值。

```
xMean[0] = 0;
xSD[0] = 1;
for (int j = 1; j <= nR; j++) {
    xMean[j] = xMean[j] / totWeights;
    if (totWeights > 1)
        xSD[j] = Math.sqrt(Math.abs(xSD[j] - totWeights * xMean[j]
                * xMean[j]) / (totWeights - 1));
    else
        xSD[j] = 0;
}
```

计算 xMean[j]的公式没有什么疑问，sum (weight[i] * x) / sum (weight[i])。xSD 的公式也很简单，忘了可以看一下 wiki，这说起来也有点矛盾，看完了论文怎么会不知道公式。

```
// Normalise input data
for (int i = 0; i < nC; i++) {
    for (int j = 0; j <= nR; j++) {
        if (xSD[j] != 0) {
            m_Data[i][j] = (m_Data[i][j] - xMean[j]) / xSD[j];
        }
    }
}
```

z-score 归范化，可以看一下 Jiawei Han 写的数据挖掘，中文版 46 页，英文版 71 页。

```
double x[] = new double[(nR + 1) * nK];
double[][] b = new double[2][x.length]; // Boundary constraints, N/A here

// Initialize
for (int p = 0; p < nK; p++) {
```

```
    int offset = p * (nR + 1);
    // Null model
    x[offset] = Math.log(sY[p] + 1.0) - Math.log(sY[nK] + 1.0);
    b[0][offset] = Double.NaN;
    b[1][offset] = Double.NaN;
    for (int q = 1; q <= nR; q++) {
        x[offset + q] = 0.0;
        b[0][offset + q] = Double.NaN;
        b[1][offset + q] = Double.NaN;
    }
}
```

数据 b 是边界约束，这里没有用，而 x 其实相当于一个二维数组，offset 第 p 个(nR+1)的位置，x[offset]是每一级的 x[0]。

```
OptEng opt = new OptEng();
opt.setDebug(m_Debug);
opt.setWeights(weights);
opt.setClassLabels(Y);

if (m_MaxIts == -1) { // Search until convergence
    x = opt.findArgmin(x, b);
    while (x == null) {
        x = opt.getVarbValues();
        if (m_Debug)
            System.out.println("200 iterations finished, not enough!");
        x = opt.findArgmin(x, b);
    }
    if (m_Debug)
        System.out.println(" -------------<Converged>--------------");
} else {
    opt.setMaxIteration(m_MaxIts);
    x = opt.findArgmin(x, b);
    if (x == null) // Not enough, but use the current value
        x = opt.getVarbValues();
}
```

m_MaxIts 是最多迭代多少次，如果它为-1 就一直迭代到收敛，opt.findArgmin，很可笑的是我导师最擅长的最优化，我却没有学到过什么。它的代码太长了，而且说的参考资料 Practical Optimization 图书馆也没有，并且那代码长的实在惊人。前面的注释上提到：In order to find the matrix B for which L is minimised, a Quasi-Newton Method is used to search for the optimized values of the m*(k-1) variables. Note that before we use the optimization procedure, we "squeeze" the matrix B into a m*(k-1) vector. For details of the optimization procedure, please check weka.core.Optimization class. 这里最优化用的是 Quasi-Newton 方法，它与 Newton 法一样，都是函数的局部最大最小值的方法。

在 distributionForInstance ：

```
public double[] distributionForInstance(Instance instance) throws
Exception {

    m_ReplaceMissingValues.input(instance);
    instance = m_ReplaceMissingValues.output();
    m_AttFilter.input(instance);
    instance = m_AttFilter.output();
    m_NominalToBinary.input(instance);
    instance = m_NominalToBinary.output();

    // Extract the predictor columns into an array
```

```
    double[] instDat = new double[m_NumPredictors + 1];
    int j = 1;
    instDat[0] = 1;
    for (int k = 0; k <= m_NumPredictors; k++) {
        if (k != m_ClassIndex) {
            instDat[j++] = instance.value(k);
        }
    }

    double[] distribution = evaluateProbability(instDat);
    return distribution;
}
```

前面的处理是与 buildClassifier 中一样的，instDat 也是第 1 个元素为 1，用剩下的元素记录属性值。evaluateProbability 的代码如下：

```
private double[] evaluateProbability(double[] data) {
    double[] prob = new double[m_NumClasses],
         v = new double[m_NumClasses];

    // Log-posterior before normalizing
    for (int j = 0; j < m_NumClasses - 1; j++) {
        for (int k = 0; k <= m_NumPredictors; k++) {
            v[j] += m_Par[k][j] * data[k];
        }
    }
    v[m_NumClasses - 1] = 0;

    // Do so to avoid scaling problems
    for (int m = 0; m < m_NumClasses; m++) {
        double sum = 0;
        for (int n = 0; n < m_NumClasses - 1; n++)
            sum += Math.exp(v[n] - v[m]);
        prob[m] = 1 / (sum + Math.exp(-v[m]));
    }

    return prob;
}
```

Product(Theta*X)，取对数后为 sum(Theta*X)，然后求每一个类别的概率可以看到下面列出来的注释，或者可以看一下 Tom Mitchell 的 Generative and discriminative classifiers: naïve bayes and logistic regression 的 13 页，公式是一样的。而这里的 sum += Math.exp(v[n]-v[m]) 这种写法是

```
The probability for class j except the last class is
 * Pj(Xi) = exp(XiBj)/((sum[j=1..(k-1)]exp(Xi*Bj))+1)
 * The last class has probability <br>
 * 1-(sum[j=1..(k-1)]Pj(Xi)) = 1/((sum[j=1..(k-1)]exp(Xi*Bj))+1)
 *
 * The (negative) multinomial log-likelihood is thus:
 * L = -sum[i=1..n]{
 * sum[j=1..(k-1)](Yij * ln(Pj(Xi))) +
 * (1 - (sum[j=1..(k-1)]Yij)) * ln(1 - sum[j=1..(k-1)]Pj(Xi))
 * } + ridge * (B^2)
```