Weka[27] SMO 源代码分析

作者：Koala++/屈伟

前两天思维混乱的状态下，把 SMO 的代码胡乱看了一下，里面的东西没有校对，错误是一定有的，现在请不要转载，把错误扩散，贴出来的目的是，如果有什么不清楚，或是不对的，希望大家能告诉我，如果我还有实力搞懂，我会努力纠正的。

还是从 buildClassifier 开始：

```java
if (!m_checksTurnedOff) {
    if (insts.checkForStringAttributes()) {
        throw new UnsupportedAttributeTypeException(
                "Cannot handle string attributes!");
    }
    if (insts.classAttribute().isNumeric()) {
        throw new UnsupportedClassTypeException(
                "SMO can't handle a numeric class! Use"
                        + "SMOreg for performing regression.");
    }
    insts = new Instances(insts);
    insts.deleteWithMissingClass();
    if (insts.numInstances() == 0) {
        throw new Exception(
                "No training instances without a missing class!");
    }

    /* Removes all the instances with weight equal to 0.
     MUST be done since condition (8) of Keerthi's paper
     is made with the assertion Ci > 0 (See equation (3a). */
    Instances data = new Instances(insts, insts.numInstances());
    for (int i = 0; i < insts.numInstances(); i++) {
        if (insts.instance(i).weight() > 0)
            data.add(insts.instance(i));
    }
    if (data.numInstances() == 0) {
        throw new Exception(
                "No training instances left after removing "
                        + "instance with either a weight "
                        + "null or a missing class!");
    }
    insts = data;
}
```

都比较简单，第一个检查是不是有 String 类型的属性，第二个是检查类别是不是数值型的，数值型的类别要用 SMOreg 进行回归分析，下面是删除没有类别的样本，再看是不是样

本数为 0，下面的注释写到，删除所有样本权重为 0 的样本，因为 Keerthi 的论文中条件(8)中有要求。再判断一次是不是样本数为 0。

```
m_onlyNumeric = true;
if (!m_checksTurnedOff) {
    for (int i = 0; i < insts.numAttributes(); i++) {
        if (i != insts.classIndex()) {
            if (!insts.attribute(i).isNumeric()) {
                m_onlyNumeric = false;
                break;
            }
        }
    }
}
```

判断是否只有数值型属性。

```
if (!m_checksTurnedOff) {
    m_Missing = new ReplaceMissingValues();
    m_Missing.setInputFormat(insts);
    insts = Filter.useFilter(insts, m_Missing);
} else {
    m_Missing = null;
}
```

ReplaceMissingValues 的注释中写到，用数据集的训练集中的众数和平均值替换所有离散和数值型属性（Replaces all missing values for nominal and numeric attributes in a dataset with the modes and means from the training data）.

```
if (!m_onlyNumeric) {
    m_NominalToBinary = new NominalToBinary();
    m_NominalToBinary.setInputFormat(insts);
    insts = Filter.useFilter(insts, m_NominalToBinary);
} else {
    m_NominalToBinary = null;
}
```

NominalToBinary 的注释中写到，转换所有的离散属性到二值的数值属性，一个属性有 K 个值会被转换成为 K 个二值属性。Converts all nominal attributes into binary numeric attributes. An attribute with k values is transformed into k binary attributes (using the one-attribute-per-value approach). Binary attributes are left binary.

```
if (m_filterType == FILTER_STANDARDIZE) {
    m_Filter = new Standardize();
    m_Filter.setInputFormat(insts);
    insts = Filter.useFilter(insts, m_Filter);
} else if (m_filterType == FILTER_NORMALIZE) {
    m_Filter = new Normalize();
    m_Filter.setInputFormat(insts);
    insts = Filter.useFilter(insts, m_Filter);
} else {
    m_Filter = null;
}
```

Standardize 的注释写到，将指定数据集中的所有数值属性都标准化为有均值 0，单位方差（Standardizes all numeric attributes in the given dataset to have zero mean and unit variance）。Nomalize 的注释写到，正规化所有的数值型值，结果值在[0,1]区间。(Normalizes all numeric values in the given dataset. The resulting values are in [0,1] for the data used to compute the normalization intervals.)

```
m_classIndex = insts.classIndex();
m_classAttribute = insts.classAttribute();

// Generate subsets representing each class
Instances[] subsets = new Instances[insts.numClasses()];
for (int i = 0; i < insts.numClasses(); i++) {
    subsets[i] = new Instances(insts, insts.numInstances());
}
for (int j = 0; j < insts.numInstances(); j++) {
    Instance inst = insts.instance(j);
    subsets[(int) inst.classValue()].add(inst);
}
for (int i = 0; i < insts.numClasses(); i++) {
    subsets[i].compactify();
}
```

得到类别索引 m_classIndex 和类别属性 m_classAttribute。Subsets 是将不同类别值的样本分开，add 函数是把样本加到相应的类别值子集中。最后的 compactify 只是为了节约空间，因为开始分配空间的时候是分配的 insts.numInstances()的大小。

```
// Build the binary classifiers
Random rand = new Random(m_randomSeed);
m_classifiers = new BinarySMO[insts.numClasses()][insts.numClasses()];
for (int i = 0; i < insts.numClasses(); i++) {
    for (int j = i + 1; j < insts.numClasses(); j++) {
        m_classifiers[i][j] = new BinarySMO();
        Instances data = new Instances(insts, insts.numInstances());
        for (int k = 0; k < subsets[i].numInstances(); k++) {
            data.add(subsets[i].instance(k));
        }
        for (int k = 0; k < subsets[j].numInstances(); k++) {
            data.add(subsets[j].instance(k));
        }
        data.compactify();
        data.randomize(rand);
        m_classifiers[i][j].buildClassifier(data, i, j,
              m_fitLogisticModels, m_numFolds, m_randomSeed);
    }
}
```

m_classifiers 是一个两维的 BinarySMO 分类器数组，这里我们可以看到，它所用的方式很落后，这也是它速度慢的一个原因。

```
// Store the sum of weights
m_sumOfWeights = insts.sumOfWeights();
```

```java
// Set class values
m_class = new double[insts.numInstances()];
m_iUp = -1;
m_iLow = -1;
for (int i = 0; i < m_class.length; i++) {
    if ((int) insts.instance(i).classValue() == cl1) {
        m_class[i] = -1;
        m_iLow = i;
    } else if ((int) insts.instance(i).classValue() == cl2) {
        m_class[i] = 1;
        m_iUp = i;
    } else {
        throw new Exception("This should never happen!");
    }
}
```

计算权重总和 m_sumOfWeights，将两个离散的类别值转换成{-1,1}。并且记录两个类别样本的最大索引值。

```java
// Check whether one or both classes are missing
if ((m_iUp == -1) || (m_iLow == -1)) {
    if (m_iUp != -1) {
        m_b = -1;
    } else if (m_iLow != -1) {
        m_b = 1;
    } else {
        m_class = null;
        return;
    }
    if (!m_useRBF && m_exponent == 1.0) {
        m_sparseWeights = new double[0];
        m_sparseIndices = new int[0];
        m_class = null;
    } else {
        m_supportVectors = new SMOset(0);
        m_alpha = new double[0];
        m_class = new double[0];
    }

    // Fit sigmoid if requested
    if (fitLogistic) {
        fitLogistic(insts, cl1, cl2, numFolds, new Random(
                randomSeed));
    }
    return;
}
```

如果 m_iUp==-1 表示第 1 种类别没有相应的样本，m_iLow==-1 表示第 2 种类别没有相应的样本，m_b 就是公式中的 b，如果两个都没有相应的样本那 m_class=nulll，最后的 fitLogistic 最用 logistic regresstion model，这个就有点远了，略过。

```
// Set the reference to the data
m_data = insts;

// If machine is linear, reserve space for weights
if (!m_useRBF && m_exponent == 1.0) {
   m_weights = new double[m_data.numAttributes()];
} else {
   m_weights = null;
}

// Initialize alpha array to zero
m_alpha = new double[m_data.numInstances()];
```

m_data 指向 insts，如果是线性的支持向量机，为 weights 保留空间，初始化 m_alpha。

```
// Initialize sets
m_supportVectors = new SMOset(m_data.numInstances());
m_I0 = new SMOset(m_data.numInstances());
m_I1 = new SMOset(m_data.numInstances());
m_I2 = new SMOset(m_data.numInstances());
m_I3 = new SMOset(m_data.numInstances());
m_I4 = new SMOset(m_data.numInstances());
```

SMOset 的构造函数如下：

```
public SMOset(int size) {

   m_indicators = new boolean[size];
   m_next = new int[size];
   m_previous = new int[size];
   m_number = 0;
   m_first = -1;
}
```

也没什么特别的。

```
// Clean out some instance variables
m_sparseWeights = null;
m_sparseIndices = null;

// Initialize error cache
m_errors = new double[m_data.numInstances()];
m_errors[m_iLow] = 1;
m_errors[m_iUp] = -1;

// Initialize kernel
if (m_useRBF) {
    m_kernel = new RBFKernel(m_data, m_cacheSize, m_gamma);
} else {
    if (m_featureSpaceNormalization) {
        m_kernel = new NormalizedPolyKernel(m_data, m_cacheSize,
                m_exponent, m_lowerOrder);
    } else {
        m_kernel = new PolyKernel(m_data, m_cacheSize, m_exponent,
                m_lowerOrder);
    }
}
```

初始化 error cache，初始化 kernel。RBF 核，多项式核，这些代码就先不管了。

```
// Build up I1 and I4
for (int i = 0; i < m_class.length; i++) {
    if (m_class[i] == 1) {
        m_I1.insert(i);
    } else {
        m_I4.insert(i);
    }
}
```

初始化 m_I1 和 m_I4，这个可以看 Keerthi 论文的第 639 页的定义

```
// Loop to find all the support vectors
int numChanged = 0;
boolean examineAll = true;
while ((numChanged > 0) || examineAll) {
    numChanged = 0;
    if (examineAll) {
        for (int i = 0; i < m_alpha.length; i++) {
            if (examineExample(i)) {
                numChanged++;
            }
        }
    } else {
        // This code implements Modification 1 from Keerthi et al.'s paper
        for (int i = 0; i < m_alpha.length; i++) {
            if ((m_alpha[i] > 0)
                    && (m_alpha[i] < m_C * m_data.instance(i).weight())) {
                if (examineExample(i)) {
                    numChanged++;
                }

                // Is optimality on unbound vectors obtained?
                if (m_bUp > m_bLow - 2 * m_tol) {
                    numChanged = 0;
                    break;
                }
            }
        }
    }

    if (examineAll) {
        examineAll = false;
    } else if (numChanged == 0) {
        examineAll = true;
    }
}
```

这段代码与 Platt 论文中第 52－53 页的伪代码基本是一样的,只有注释的地方是 Keerthi 的改进。可以看 Platt 论文中的第 47 页，原文如下：To speed training, the outer loop does not always iterate through the entire training set. After one pass through the training set, the outer loop iterates over only those examples whose Lagrange multipliers are neither 0 nor C。Again, each example is checked against the KKT conditions, and violating examples are eligible for immediate optimization and update. The outer loop makes repeated passes over the non-bound examples until all of the non-bound examples obey the KKT conditions within ethta. The outer loop then iterates over the entire set again. The outer loop keeps alternating between single passes over the entire training set and multiple passes over the non-bound subset until the entire training set obeys the KKT conditions within ethta. At that point, the algorithm terminates.

这里的 examineAll 就是论文中说是是否需要 iterate through the entire training set。而 examineExample 就是判断这个样本是不是在边界上，也就是它的 Lagrange 乘子不是 0 也不是 C。而 else 的代码第 1 个 if 与 Platt 给出的伪代码是一样的,看边界上的样本是否满足 KKT 条件了，Is optimality on unbounded vectors obtained？这句话对应的是 Keerthi 中的公式 2.8。while 中的 numChanged > 0 表示如果还有改变的，那么就继续。如果 entire training set obeys the KKT conditions within ethta. At that point the algorithm terminates。

接下来是 examineExample 的代码：

```
double y2, alph2, F2;
int i1 = -1;

y2 = m_class[i2];
alph2 = m_alpha[i2];
if (m_I0.contains(i2)) {
    F2 = m_errors[i2];
} else {
    F2 = SVMOutput(i2, m_data.instance(i2)) + m_b - y2;
    m_errors[i2] = F2;

    // Update thresholds
    if ((m_I1.contains(i2) || m_I2.contains(i2)) && (F2 < m_bUp)) {
        m_bUp = F2;
        m_iUp = i2;
    } else if ((m_I3.contains(i2) || m_I4.contains(i2))
           && (F2 > m_bLow)) {
        m_bLow = F2;
        m_iLow = i2;
    }
}
```

y2 是第 i2 个样本的类别，而 alph2 是第 i2 个 alpha 值，如果 i2 是非边界点那么它 F2 就在 error cache 中，直接取得就可以了。而 else 则是不在边界上的点，当然也就不在 error cache 中了，关于 SVMOutput 的代码马上再看，而下面的 if 和 else if 则是 Keerthi 中的公式 2.4。更新相应的阈值。

SVMOutput 中 RBF 核的代码就不看了，也就是只看 if 的代码：

```
double result = 0;

// Is the machine linear?
if (!m_useRBF && m_exponent == 1.0) {

    // Is weight vector stored in sparse format?
    if (m_sparseWeights == null) {
        int n1 = inst.numValues();
        for (int p = 0; p < n1; p++) {
            if (inst.index(p) != m_classIndex) {
                result += m_weights[inst.index(p)]
                        * inst.valueSparse(p);
            }
        }
    } else {
        int n1 = inst.numValues();
        int n2 = m_sparseWeights.length;
        for (int p1 = 0, p2 = 0; p1 < n1 && p2 < n2;) {
            int ind1 = inst.index(p1);
            int ind2 = m_sparseIndices[p2];
            if (ind1 == ind2) {
                if (ind1 != m_classIndex) {
                    result += inst.valueSparse(p1)
                            * m_sparseWeights[p2];
                }
                p1++;
                p2++;
            } else if (ind1 > ind2) {
                p2++;
            } else {
                p1++;
            }
        }
    }
}
result -= m_b;

return result;
```

如果不想看稀疏矩阵的压缩存储方式，看最上面的那一点就可以了，其中的 for 循环就是 w 和 x 的点积，最后减去阈值。而这也就是刚才为什么计算 F2 的时候要-m_b 的原因。公式是 Keerthi 论文 639 页。

```
// Check optimality using current bLow and bUp and, if
// violated, find an index i1 to do joint optimization
// with i2...
boolean optimal = true;
if (m_I0.contains(i2) || m_I1.contains(i2) || m_I2.contains(i2)) {
    if (m_bLow - F2 > 2 * m_tol) {
        optimal = false;
        i1 = m_iLow;
    }
}
if (m_I0.contains(i2) || m_I3.contains(i2) || m_I4.contains(i2)) {
    if (F2 - m_bUp > 2 * m_tol) {
        optimal = false;
        i1 = m_iUp;
    }
}
if (optimal) {
    return false;
}
```

注释上写到，用当前的 bLow 和 bUp 来检查最优性，如果违反了，找 i1 垰与 i2 联合最优化。可以从 Keerthi 中的 2.5 中看到 m_bLow 和 m_bUp 分别是 Fi 在相应集合中的最小值和最大值，所有如果是满足最优性条件，不应该出现代码中的两个 if 如果出现了，则说明不满足，如果已经满足就返回 false。而将 i1 赋以 m_iLow 或是 m_iUp，因为它能使 progress 进展最快。

```
// For i2 unbound choose the better i1...
if (m_I0.contains(i2)) {
    if (m_bLow - F2 > F2 - m_bUp) {
        i1 = m_iLow;
    } else {
        i1 = m_iUp;
    }
}
if (i1 == -1) {
    throw new Exception("This should never happen!");
}
return takeStep(i1, i2, F2);
```

m_bLow-F2 与 F2-m_bUp 必有一个在 m_tol 范围内是正值，也就是 m_bLow 本应该比

F2 小，但是如果这里比它大了，那么又因为 m_bLow<m_bUp(公式 2.6)，那么 F2-m_bUp 应该是一个负值，而相反如果 F2>m_bUp 刚好相反。

下面是 takeStep 的代码：

```
double C1 = m_C * m_data.instance(i1).weight();
double C2 = m_C * m_data.instance(i2).weight();

// Don't do anything if the two instances are the same
if (i1 == i2) {
    return false;
}

// Initialize variables
alph1 = m_alpha[i1];
alph2 = m_alpha[i2];
y1 = m_class[i1];
y2 = m_class[i2];
F1 = m_errors[i1];
s = y1 * y2;
```

从这里可以看出来一点，为什么前面说不能把样本的权重设为 0。惩罚因子为一个常量乘以样本的权重。下面注释写到如果两个样本一样就什么也不做。再下面是初始化，s=y1 * y2 的说明在 Platt 论文的 45 页最下面图的说明中。

```
// Find the constraints on a2
if (y1 != y2) {
    L = Math.max(0, alph2 - alph1);
    H = Math.min(C2, C1 + alph2 - alph1);
} else {
    L = Math.max(0, alph1 + alph2 - C1);
    H = Math.min(C2, alph1 + alph2);
}
if (L >= H) {
    return false;
}
```

Platt 的公式 12.3 和 12.4 开始我看这个公式竟然有点糊涂，如果你也不幸糊涂了，把那个点做到 x 轴和 y 轴的垂线就明白了。

```
// Compute second derivative of objective function
k11 = m_kernel.eval(i1, i1, m_data.instance(i1));
k12 = m_kernel.eval(i1, i2, m_data.instance(i1));
k22 = m_kernel.eval(i2, i2, m_data.instance(i2));
eta = 2 * k12 - k11 - k22;
```

这段代码是 Platt 公式 12.5，m_kernel.eval 过会再看。

```
// Check if second derivative is negative
if (eta < 0) {

    // Compute unconstrained maximum
    a2 = alph2 - y2 * (F1 - F2) / eta;

    // Compute constrained maximum
    if (a2 < L) {
        a2 = L;
    } else if (a2 > H) {
        a2 = H;
    }
}
```

注意一下 Platt 论文中的这段话：Under normal circumstances, there will be a maximum along the direction of the linear equality constraint, and eta will be less than zero, In this case, SMO computes the maximum along the direction of the constraint，下面是公式 12.6 和 12.7。就是上面代码的解释。

```
else {
    // Look at endpoints of diagonal
    f1 = SVMOutput(i1, m_data.instance(i1));
    f2 = SVMOutput(i2, m_data.instance(i2));
    v1 = f1 + m_b - y1 * alph1 * k11 - y2 * alph2 * k12;
    v2 = f2 + m_b - y1 * alph1 * k12 - y2 * alph2 * k22;
    double gamma = alph1 + s * alph2;
    Lobj = (gamma - s * L) + L - 0.5 * k11 * (gamma - s * L)
           * (gamma - s * L) - 0.5 * k22 * L * L - s * k12
           * (gamma - s * L) * L - y1 * (gamma - s * L) * v1 - y2
           * L * v2;
    Hobj = (gamma - s * H) + H - 0.5 * k11 * (gamma - s * H)
           * (gamma - s * H) - 0.5 * k22 * H * H - s * k12
           * (gamma - s * H) * H - y1 * (gamma - s * H) * v1 - y2
           * H * v2;
    if (Lobj > Hobj + m_eps) {
        a2 = L;
    } else if (Lobj < Hobj - m_eps) {
        a2 = H;
    } else {
        a2 = alph2;
    }
}
if (Math.abs(a2 - alph2) < m_eps * (a2 + alph2 + m_eps)) {
    return false;
}
```

这段代码的解释在 Platt 论文的 47 页。In any event, SMO will work even when eta is not negative, in which case the objective function W should be evaluated at each end of the line segment. Only those terms in the objective function that depend on alpha_2 need be evaluated (see equation (12.23)). SMO moves the Lagrange multipliers to the end point with the highest value of the objective function. If the objective function is the same at the both ends and the kernel obeys

Mercer’s conditions, then the joint maximization cannot make progress.

Gamma 的计算公式 Platt 论文的（12.22），Lobj 和 Hobj 就是论文提到的 W should be evaluated at each end of the line segment.公式是 12.23。下面的 if / else if/ else 则对应 SMO moves the Lagrane multipliers to the end point with the highest value of the objective function. 最后的一个 if 对应 If the objective function is the same at the both ends and the kernel obeys Mercer’s conditions, then the joint maximization cannot make progress.

```
// To prevent precision problems
if (a2 > C2 - m_Del * C2) {
    a2 = C2;
} else if (a2 <= m_Del * C2) {
    a2 = 0;
}
```

精度问题处理代码，m_Del 是一个非常小的数，看 a2 是不是已经可以认为是 C2 或是 0 了。

```
// Recompute a1
a1 = alph1 + s * (alph2 - a2);

// To prevent precision problems
if (a1 > C1 - m_Del * C1) {
    a1 = C1;
} else if (a1 <= m_Del * C1) {
    a1 = 0;
}
```

Platt 的公式 12.8，和精度问题的处理。

```
// Update sets
if (a1 > 0) {
    m_supportVectors.insert(i1);
} else {
    m_supportVectors.delete(i1);
}
if ((a1 > 0) && (a1 < C1)) {
    m_I0.insert(i1);
} else {
    m_I0.delete(i1);
}
if ((y1 == 1) && (a1 == 0)) {
    m_I1.insert(i1);
} else {
    m_I1.delete(i1);
}
if ((y1 == -1) && (a1 == C1)) {
    m_I2.insert(i1);
} else {
    m_I2.delete(i1);
}
if ((y1 == 1) && (a1 == C1)) {
    m_I3.insert(i1);
} else {
    m_I3.delete(i1);
}
if ((y1 == -1) && (a1 == 0)) {
    m_I4.insert(i1);
} else {
    m_I4.delete(i1);
}
if (a2 > 0) {
    m_supportVectors.insert(i2);
} else {
    m_supportVectors.delete(i2);
}
if ((a2 > 0) && (a2 < C2)) {
    m_I0.insert(i2);
} else {
    m_I0.delete(i2);
}
if ((y2 == 1) && (a2 == 0)) {
    m_I1.insert(i2);
} else {
    m_I1.delete(i2);
}
if ((y2 == -1) && (a2 == C2)) {
    m_I2.insert(i2);
} else {
    m_I2.delete(i2);
}
if ((y2 == 1) && (a2 == C2)) {
    m_I3.insert(i2);
} else {
    m_I3.delete(i2);
}
if ((y2 == -1) && (a2 == 0)) {
    m_I4.insert(i2);
} else {
    m_I4.delete(i2);
}
```

m_supportVectors，m_I0，m_I1，m_I2，m_I3，m_I4 是加入 i1，i2 还是删除 i1，i2。看 Keerthi 论文的 639 对它们的定义，m_supprotVectors 是支持向量索引数组，就是乘子大于 0 的样本索引。

```
// Update weight vector to reflect change a1 and a2, if linear SVM
if (!m_useRBF && m_exponent == 1.0) {
    Instance inst1 = m_data.instance(i1);
    for (int p1 = 0; p1 < inst1.numValues(); p1++) {
        if (inst1.index(p1) != m_data.classIndex()) {
            m_weights[inst1.index(p1)] += y1 * (a1 - alph1)
                    * inst1.valueSparse(p1);
        }
    }
    Instance inst2 = m_data.instance(i2);
    for (int p2 = 0; p2 < inst2.numValues(); p2++) {
        if (inst2.index(p2) != m_data.classIndex()) {
            m_weights[inst2.index(p2)] += y2 * (a2 - alph2)
                    * inst2.valueSparse(p2);
        }
    }
}
```

更新权重向量，Platt 的公式 12.12。

```
// Update error cache using new Lagrange multipliers
for (int j = m_I0.getNext(-1); j != -1; j = m_I0.getNext(j)) {
    if ((j != i1) && (j != i2)) {
        m_errors[j] += y1 * (a1 - alph1)
                * m_kernel.eval(i1, j, m_data.instance(i1)) + y2
                * (a2 - alph2)
                * m_kernel.eval(i2, j, m_data.instance(i2));
    }
}
```

Platt 的公式 12.11。

```
// Update error cache for i1 and i2
m_errors[i1] += y1 * (a1 - alph1) * k11 + y2 * (a2 - alph2) * k12;
m_errors[i2] += y1 * (a1 - alph1) * k12 + y2 * (a2 - alph2) * k22;
```

还是公式 12.11，只是 k11，k12，k22 没有必要再次计算。

```
// Update array with Lagrange multipliers
m_alpha[i1] = a1;
m_alpha[i2] = a2;
```

更新 Lagrange 乘子数组。

```
// Update thresholds
m_bLow = -Double.MAX_VALUE;
m_bUp = Double.MAX_VALUE;
m_iLow = -1;
m_iUp = -1;
for (int j = m_I0.getNext(-1); j != -1; j = m_I0.getNext(j)) {
    if (m_errors[j] < m_bUp) {
        m_bUp = m_errors[j];
        m_iUp = j;
    }
    if (m_errors[j] > m_bLow) {
        m_bLow = m_errors[j];
        m_iLow = j;
    }
}
if (!m_I0.contains(i1)) {
    if (m_I3.contains(i1) || m_I4.contains(i1)) {
        if (m_errors[i1] > m_bLow) {
            m_bLow = m_errors[i1];
            m_iLow = i1;
        }
    } else {
        if (m_errors[i1] < m_bUp) {
            m_bUp = m_errors[i1];
            m_iUp = i1;
        }
    }
}
if (!m_I0.contains(i2)) {
    if (m_I3.contains(i2) || m_I4.contains(i2)) {
        if (m_errors[i2] > m_bLow) {
            m_bLow = m_errors[i2];
            m_iLow = i2;
        }
    } else {
        if (m_errors[i2] < m_bUp) {
            m_bUp = m_errors[i2];
            m_iUp = i2;
        }
    }
}
if ((m_iLow == -1) || (m_iUp == -1)) {
    throw new Exception("This should never happen!");
}
```

先在 m_I0 找看有没有在[m_bLow,m_bUp]之外的值，有则更新，然后在 m_I3，m_I4，m_I1，m_I2 中对 i1 和 i2 索引的 m_errors 判断。

回到 BinarySMO 中的 buildClassifier：

```
// Set threshold
m_b = (m_bLow + m_bUp) / 2.0;

// Save memory
m_kernel.clean();
```

```
m_errors = null;
m_I0 = m_I1 = m_I2 = m_I3 = m_I4 = null;
```

设置阈值，释放空间。

```
// If machine is linear, delete training data
// and store weight vector in sparse format
if (!m_useRBF && m_exponent == 1.0) {

    // We don't need to store the set of support vectors
    m_supportVectors = null;

    // We don't need to store the class values either
    m_class = null;

    // Clean out training data
    if (!m_checksTurnedOff) {
        m_data = new Instances(m_data, 0);
    } else {
        m_data = null;
    }

    // Convert weight vector
    double[] sparseWeights = new double[m_weights.length];
    int[] sparseIndices = new int[m_weights.length];
    int counter = 0;
    for (int i = 0; i < m_weights.length; i++) {
        if (m_weights[i] != 0.0) {
            sparseWeights[counter] = m_weights[i];
            sparseIndices[counter] = i;
            counter++;
        }
    }
    m_sparseWeights = new double[counter];
    m_sparseIndices = new int[counter];
    System.arraycopy(sparseWeights, 0, m_sparseWeights, 0, counter);
    System.arraycopy(sparseIndices, 0, m_sparseIndices, 0, counter);

    // Clean out weight vector
    m_weights = null;

    // We don't need the alphas in the linear case
    m_alpha = null;
}
```

释放空间的代码，和给 m_sparseWeights 和 m_sparseIndices 赋值的代码。

下面是 distridutionForInstance 的代码：

```java
// Filter instance
if (!m_checksTurnedOff) {
    m_Missing.input(inst);
    m_Missing.batchFinished();
    inst = m_Missing.output();
}

if (!m_onlyNumeric) {
    m_NominalToBinary.input(inst);
    m_NominalToBinary.batchFinished();
    inst = m_NominalToBinary.output();
}

if (m_Filter != null) {
    m_Filter.input(inst);
    m_Filter.batchFinished();
    inst = m_Filter.output();
}
```

buildClassifier 中已经看过，略过。这里只看 m_fitLogisticModels 的代码：

```java
if (!m_fitLogisticModels) {
    double[] result = new double[inst.numClasses()];
    for (int i = 0; i < inst.numClasses(); i++) {
        for (int j = i + 1; j < inst.numClasses(); j++) {
            if ((m_classifiers[i][j].m_alpha != null)
                    || (m_classifiers[i][j].m_sparseWeights != null)) {
                double output = m_classifiers[i][j].SVMOutput(-1, inst);
                if (output > 0) {
                    result[j] += 1;
                } else {
                    result[i] += 1;
                }
            }
        }
    }
    Utils.normalize(result);
    return result;
}
```

在 m_fitLogisticModels 为 false 的情况下 SVMOutput 第一个参数没有意义，这里是一个投票的过程，如果这次 i 和 j 类别分类时，分到了 j 中 result[j]++;，否则 i++;

```java
/**
 * The polynomial kernel : K(x, y) = <x, y>^p or K(x, y) = ( <x, y>+1)^p
*/
public class PolyKernel extends CachedKernel {
    /** Use lower-order terms? */
    private boolean m_lowerOrder = false;

    /** The exponent for the polynomial kernel. */
    private double m_exponent = 1.0;

    /**
     * Creates a new <code>PolyKernel</code> instance.
     *
     * @param dataset
     *           the training dataset used.
     * @param cacheSize
     *           the size of the cache (a prime number)
     */
    public PolyKernel(Instances dataset, int cacheSize, double exponent,
           boolean lowerOrder) {

        super(dataset, cacheSize);

        m_exponent = exponent;
        m_lowerOrder = lowerOrder;
        m_data = dataset;
    }

    protected double evaluate(int id1, int id2, Instance inst1)
           throws Exception {

        double result;
        if (id1 == id2) {
            result = dotProd(inst1, inst1);
        } else {
            result = dotProd(inst1, m_data.instance(id2));
        }
        // Use lower order terms?
        if (m_lowerOrder) {
            result += 1.0;
        }
        if (m_exponent != 1.0) {
            result = Math.pow(result, m_exponent);
        }
        return result;
    }
}
```

m_lowerOrder 是指是 K(x,y)=<x,y>^p 还是 K(x,y)=(<x,y>+1)^p，m_exponent 不用说当然是指数 p 的值。

```
protected final double dotProd(Instance inst1, Instance inst2)
        throws Exception {

    double result = 0;

    // we can do a fast dot product
    int n1 = inst1.numValues();
    int n2 = inst2.numValues();
    int classIndex = m_data.classIndex();
    for (int p1 = 0, p2 = 0; p1 < n1 && p2 < n2;) {
        int ind1 = inst1.index(p1);
        int ind2 = inst2.index(p2);
        if (ind1 == ind2) {
            if (ind1 != classIndex) {
                result += inst1.valueSparse(p1) * inst2.valueSparse(p2);
            }
            p1++;
            p2++;
        } else if (ind1 > ind2) {
            p2++;
        } else {
            p1++;
        }
    }
    return (result);
}
```

这个内积的计算，就是 Platt 论文 50 页中的伪代码。

```
// we can only cache if we know the indexes
if (id1 >= 0) {

    // Use full cache?
    if (m_cacheSize == 0) {
        if (m_kernelMatrix == null) {
            m_kernelMatrix = new double[m_data.numInstances()][];
            for (int i = 0; i < m_data.numInstances(); i++) {
                m_kernelMatrix[i] = new double[i + 1];
                for (int j = 0; j <= i; j++) {
                    m_kernelEvals++;
                    m_kernelMatrix[i][j] = evaluate(i, j, m_data
                            .instance(i));
                }
            }
        }
        m_cacheHits++;
        result = (id1 > id2) ? m_kernelMatrix[id1][id2]
                : m_kernelMatrix[id2][id1];
        return result;
    }
```

注释第一行，我们只缓存我们知道的 indexes，下面是不是全部缓存，如果还没有缓存过，那么分配空间，m_kernelMatrix 是一个三角矩阵，计算就是用刚才看过的 evaluate 函数，因为 m_kernelMatrix[id1][id2]=m_kernelMatrix[id2][id1]没有必要记录两次，只需在用的时候判断一下大小就可以了。

```
    // Use LRU cache
    if (id1 > id2) {
        key = (id1 + ((long) id2 * m_numInsts));
    } else {
        key = (id2 + ((long) id1 * m_numInsts));
    }
    location = (int) (key % m_cacheSize) * m_cacheSlots;
    int loc = location;
    for (int i = 0; i < m_cacheSlots; i++) {
        long thiskey = m_keys[loc];
        if (thiskey == 0)
            break; // empty slot, so break out of loop early
        if (thiskey == (key + 1)) {
            m_cacheHits++;
            // move entry to front of cache (LRU) by swapping
            // only if it's not already at the front of cache
            if (i > 0) {
                double tmps = m_storage[loc];
                m_storage[loc] = m_storage[location];
                m_keys[loc] = m_keys[location];
                m_storage[location] = tmps;
                m_keys[location] = thiskey;
                return tmps;
            } else
                return m_storage[loc];
        }
        loc++;
    }
}

result = evaluate(id1, id2, inst1);

m_kernelEvals++;
```

```
// store result in cache
if (key != -1) {
    // move all cache slots forward one array index
    // to make room for the new entry
    System.arraycopy(m_keys, location, m_keys, location + 1,
            m_cacheSlots - 1);
    System.arraycopy(m_storage, location, m_storage, location + 1,
            m_cacheSlots - 1);
    m_storage[location] = result;
    m_keys[location] = (key + 1);
}
return result;
```

注释提到的 LRU 算法全称是 Least Recently Used。这里先看一下它的构造函数：

```
/**
 * Initializes the kernel cache. The actual size of the cache in bytes
 * is (64 * cacheSize).
 */
protected CachedKernel(Instances data, int cacheSize) {
    m_data = data;
    m_cacheSize = cacheSize;
    if (cacheSize > 0) {

        // Use LRU cache
        m_storage = new double[m_cacheSize * m_cacheSlots];
        m_keys = new long[m_cacheSize * m_cacheSlots];
    }

    m_numInsts = m_data.numInstances();
}
```

m_storeage 和 m_keys 的大小都是 m_cacheSize*m_cacheSlots。回到 eval 中 key 的计算也是很常规的，这就相当于是把 id1 看成了一个个数位，而 id2 是个 m_numInsts 位数。接下来计算 location，如果 m_keys[loc]==0 表示这个 slot 是空的，跳出循环。如果为 key+1，那么按 LRU 算法就要交换 loc 和 location 两个下标的 m_storage 和 m_keys 的值。这个和操作系统中讲的差不多，一个 slot 的大小是 m_cacheSlots，那么也就每 m_cacheSlots 个后是另一个 key 的 slot 了。Key!=-1 下面的两个 arraycopy 是把最前面的那个位置移出来给最新的 result 和 key。