Weka[34] LWL 源代码分析

作者：Koala++/屈伟

看了笨笨的 Blog，想起来 Ng Andrew 说过他的导师 Michael Jordan(与那个篮球明星同名)最喜欢的就是 LWL 算法。

Locally weighted learning. Uses an instance-based algorithm to assign instance weights which are then used by a specified WeightedInstancesHandler. Can do classification (e.g. using naive Bayes) or regression (e.g. using linear regression)

Locally weighted learning(LWL)，用基于样本的算法用一个特定的 weightedInstancesHandler 赋给样本权重。可以进行分类或是回归。

我也大概看了一下它是如何来实现的。

它是继承自 SingleClassifierEnhancer 类，在它的构造函数中：

```
public LWL() {
    m_Classifier = new weka.classifiers.trees.DecisionStump();
}
```

分类器也是用的 DecisionStump 算法，在 Adaboost 篇里已经提过它是什么了。

接下来是 buildClassifier 函数：

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

    if (!(m_Classifier instanceof WeightedInstancesHandler)) {
        throw new IllegalArgumentException("Classifier must be a "
               + "WeightedInstancesHandler!");
    }

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

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

    // only class? -> build ZeroR model
    if (instances.numAttributes() == 1) {
        m_ZeroR = new weka.classifiers.rules.ZeroR();
        m_ZeroR.buildClassifier(instances);
        return;
    } else {
        m_ZeroR = null;
    }

    m_Train = new Instances(instances, 0, instances.numInstances());

    m_NNSearch.setInstances(m_Train);
}
```

如果分类器不能利用样本权重，那么抛出异常，后面的代码会解释这个原因的。再下面是将所有的样本都默认为是训练样本，m_NNSearch 可以看一下我(Koala++)写《Weka 开发[18]——寻找 K 个邻居》，

```
public void updateClassifier(Instance instance) throws Exception {

    if (m_Train == null) {
```

```
            throw new Exception("No training instance structure set!");
        } else if (m_Train.equalHeaders(instance.dataset()) == false) {
            throw new Exception("Incompatible instance types");
        }
        if (!instance.classIsMissing()) {
            m_NNSearch.update(instance);
            m_Train.add(instance);
        }
    }
```

LWL 实现了 UpdatableClassifier，表示它是可以增量学习的，其实也不太是那么回事感觉。m_NNSearch.update 还牵扯了一大堆代码，这里先不说了，以后看距离类的时候再看。

再下面是 distributionForInstance，把这个函数拆开来看：

```
m_NNSearch.addInstanceInfo(instance);

int k = m_Train.numInstances();
if ((!m_UseAllK && (m_kNN < k))
        && !(m_WeightKernel == INVERSE || m_WeightKernel == GAUSS)) {
    k = m_kNN;
}
```

如果不是用全部的数据，并且 k 已经大于指定的样本数，不是用下面两个加权方法，就将 k 指定为用户指定的样本数。

```
Instances neighbours = m_NNSearch.kNearestNeighbours(instance, k);
double distances[] = m_NNSearch.getDistances();

// IF LinearNN has skipped so much that <k neighbours are remaining.
if (k > distances.length)
    k = distances.length;
```

kNearestNeighbours 是得到 instance 的 k 个邻居，而 getDistances 是 Returns the distances of the k nearest neighbours. The kNearestNeighbours or nearestNeighbour needs to be called first for this to work. (返回 k 个邻居的距离。需要先调用 kNearestNeighbours 或是 nearestNeighbour)，简单地说，就是这个函数设计的不好，因为类用户要知道调用顺序。下面是如果 LinearNN 跳的过多了，那么只是小于 K 个邻居了，这里就是上面那个函数的作用。

```
// Determine the bandwidth
double bandwidth = distances[k - 1];

// Check for bandwidth zero
if (bandwidth <= 0) {
    // if the kth distance is zero than give all instances the same
    // weight
    for (int i = 0; i < distances.length; i++)
        distances[i] = 1;
} else {
    // Rescale the distances by the bandwidth
    for (int i = 0; i < distances.length; i++)
        distances[i] = distances[i] / bandwidth;
}
```

下面是就是确定范围，如果 bandwidth=0 当然就是所有的样本和要分类的样本是在一起的，那么所有的距离都是 1。否则就将它们变换到 (0,1]区间里。

```
// Pass the distances through a weighting kernel
for (int i = 0; i < distances.length; i++) {
    switch (m_WeightKernel) {
    case LINEAR:
        distances[i] = 1.0001 - distances[i];
        break;
```

```
    case EPANECHNIKOV:
        distances[i] = 3 / 4D * (1.0001 - distances[i] * distances[i]);
        break;
    case TRICUBE:
        distances[i] = Math
                .pow((1.0001 - Math.pow(distances[i], 3)), 3);
        break;
    case CONSTANT:
        // System.err.println("using constant kernel");
        distances[i] = 1;
        break;
    case INVERSE:
        distances[i] = 1.0 / (1.0 + distances[i]);
        break;
    case GAUSS:
        distances[i] = Math.exp(-distances[i] * distances[i]);
        break;
    }
}
```

将距离通过一个加权的核，这是一个很有意思的东西，可以看一下 pattern recognition and machine learning 的 2.5 节，简单地说就是 k 近邻是在空间中找目标样本的邻居，那也就是用邻居的数量，来确定了这个样本邻居所在的子空间大小，而核是在目标样本周围划出来一个子空间，用它来确定样本邻居的数量。

常用的是 GAUSS，可以去笨笨的日志中找《关于 LWL (Local weighting Learning)的一些问题》这一篇看一下。

```
// Set the weights on the training data
double sumOfWeights = 0, newSumOfWeights = 0;
for (int i = 0; i < distances.length; i++) {
    double weight = distances[i];
    Instance inst = (Instance) neighbours.instance(i);
    sumOfWeights += inst.weight();
    newSumOfWeights += inst.weight() * weight;
    inst.setWeight(inst.weight() * weight);
    // weightedTrain.add(newInst);
}

// Rescale weights
for (int i = 0; i < neighbours.numInstances(); i++) {
    Instance inst = neighbours.instance(i);
    inst.setWeight(inst.weight() * sumOfWeights / newSumOfWeights);
}

// Create a weighted classifier
m_Classifier.buildClassifier(neighbours);

// Return the classifier's predictions
return m_Classifier.distributionForInstance(instance);
```

将这些目标样本的邻居设置新的权重，再 rescale，LWL 与别的分类器的区别就是它通过邻居来训练并且考虑邻居的距离，再下来就是调用基分类器的 distributionForInstance。