MLAPP-C16

16. 自适应基函数模型

16.1 引言

在第14和第15章，我们讨论了核函数方法，它为在回归和分类中使用非线性模型提供了有力的方法。预测结果的形式为，其中我们定义：

 （16.1）

其中表示为全体训练集或者其子集。具备这种形式的模型实际上执行一种被称为**样式匹配**（template matching）的操作，它们将输入**x**与存储的原型进行对比。

尽管这种方式可能奏效，但它依赖于拥有一个好的核函数，从而衡量数据向量之间的相似度。通常情况下，遇到一个好的核函数是相当困难的。举例来说，如何定义两张图片之间的相似度？基于像素级别的强度对比（高斯核就是对应这种方法）效果并不好。尽管人工设计核函数以解决特定的任务是有可能的（实际上也是比较常见的），但如果我们可以通过学习得到核函数，那将是一件非常有趣的事情。

在章节15.2.4，我们讨论了一种通过最大化边缘似然函数方式学习核函数参数的方法。举例来说，如果我们使用ARD核，

 （16.2）

我们可以估计，也就是进行非线性特征选择。然而，这样的方法计算成本高。另一种方法，被称为复合核函数学习，它使用基本核函数的凸组合，然后估计混合权重。但是依赖于有好的基本核函数（其计算成本也很高）。

一种可选的方案是完全不使用核函数，尝试从数据中直接学习有用的特征。也就是说，我们建立所谓的**自适应基函数模型**（adaptive basis function model, ABM），其形式为：

 （16.3）

其中被称为第m个基函数，它是从数据中学习到的。这个框架涵盖我们在本章介绍的所有模型。

典型情况下，基函数是含参数的，所以我们将基函数写成，其中为基函数本身的参数。我们将使用表示模型的整个参数集合。最终的模型在参数空间中不再是一个线性模型，所以我们只能够找到一个关于的MLE或者MAP估计的局部最优解。然而，这样的模型经常明显优于线性模型，这一点我们将在后文看到。

16.2 分类和回归树

**分类和回归树**（classification and regression trees, CART）模型，又被称为**决策树**（decision trees）（不要与决策论中的决策树混淆），该模型通过对输入空间的不断分割，再在每一个形成的子区域中再定义一个局部模型。这个过程可以表示成一棵树，每一个子区域拥有一片叶子，关于这一点我们将在后面解释。

16.2.1 基本原理

为了解释CART方法，考虑图16.1（a）所示的树。第一个节点询问*x*1是否小于某个阈值*t*1。如果满足，再询问*x*2是否小于另一个阈值*t*2。如果满足，我们将到达左侧底部的空间象限*R*1。如果不满足，我们将询问*x*1是否小于某个阈值*t*3。以此类推。这些**与坐标轴平行的分割**（axis parallel splits）将2维空间划分为5个区域，如果16.1（b）所示。现在我们为每一个子区域关联一个平均响应，从而形成了如图16.1（b）所示的平面。

我们将这个模型写成如下的形式：

 （16.4）

其中*R*m表示第m个区域，*w*m为该区域的平均响应，**v**m代表了从根节点到第m个叶子节点的路径中，所有用于分割的变量以及分割的阈值。基于这些，我们可以很清晰地发现，CART模型只是一个自适应的基函数模型，其中基函数定义了子区域，权重指定了每个区域的平均响应。后面我们将讨论如何找到这些基函数。

我们将上述模型泛化到分类问题中，通过在每个叶子节点中存储响应的类别而不是平均的响应来实现这一转变。图16.2说明了这一点。该模型可以用来对图1.1中的数据进行分类，举例来说，我们首先检查目标的颜色。如果它是蓝色，我们跟随左侧的分支，并以一个标签为“4,0”的叶子节点结束，这就意味着我们有4个正例和0个负例满足这一准则。所以如果**x**是蓝色，我们预测*p*(*y*=1|**x**)=4/4。如果它是红色，我们接着检查它的形状：如果它是红色，但不是一个椭圆，我们预测*p*(*y*=1|**x**)=0/2；如果它是其他颜色，我们检查它的大小：如果小于10，我们预测*p*(*y*=1|**x**)=4/4，否则*p*(*y*=1|**x**)=0/5。这些概率值仅是满足一系列特征值组合的正例数量的经验占比，这些组合定义了一条从根部到叶子结点的路径。

16.2.2 树的生成

找到一组数据的最优分割方案是一个NP-复杂问题，所以通常使用算法16.1中展示的贪心算法来计算一个局部最优的MLE。这种方法被CART，C4.5和ID3使用，这三个算法是该方法的著名实现。

|  |
| --- |
| 算法16.1：生成分类/回归树的递归程序 |
| 1. function fitTree(node, D, depth);  2. node.prediction = mean(*y*i:i∈) // or class label distribution；  3. (*j*\*,*t*\*,L,R) =split();  4. **if** not worthSplitting(depth, cost, L, *D*R) then  5 Return node  6. **else**  7. node.test = //类似的函数  8. node.left = fitTree(node, L, depth+1)  9. node.right = fitTree(node, *D*R, depth+1)  10. return node; |

分割函数用于选择最好的特征以及该特征对应的最好的分割值，函数如下：

 （16.5）

其中损失函数的定义将在下面给出。为了符号上的简单，我们假设所有的输入都是实数或者有序的，这样特征*xij*与数值*t*的对比才有意义。特征*j*可取的阈值集合*j*可以通过对*xij*的所有可取值进行排序后获得。举例来说，如果特征1的可取值为{4.5，-12，72，-12}，那么我们设置1={-12，4.5，72}。如果输入是类别变量，最常用的方法是对每个可取的类别标签*ck*，采用*xij* = *ck*和*xij* ≠ *ck*的方式进行分割。尽管我们允许多个分支（即非二叉树），但这会导致**数据碎片**（data fragmentation），意味着每一棵子树中可能只有很少的数据，进而导致过拟合。

用于判断一个节点是否值得分割的函数可以基于以下几种中止原则：

* 继续分割所导致的损失的降低是否足够小？典型情况下，我们定义基于某个特征进行分割所引起的增益为损失降低幅度的规范化测度：

 （16.6）

* 树的深度是否已经超过了最大的设置值；
* 在L或者R中的响应的分布是否足够统一（比如，所有的标签都一样，那么这个分布就是纯（pure）的）？
* 在L或者R中的样本数量是否太少？

接下来所有的内容都是确定损失函数，从而评估候选分割点的质量。这取决于我们的目标是分类还是回归。我们在下面讨论两种情况。

16.2.2.1 回归损失

在回归的情况下，我们定义损失为：

 （16.7）

其中为在指定数据集中响应变量的均值。另一种可选的方案是，为每个叶子节点训练一个线性回归模型，每个模型使用从根节点到相应叶子节点的路径上的变量作为模型特征，然后以残差作为衡量标准。

16.2.2.2 分类损失

在分类问题中有几种衡量分割效果的方法。首先，我们针对在叶子节点中满足*Xj* < *t*的数据训练一个multinoulli模型，训练的方式是估计类条件概率：

 （16.8）

其中为叶子节点中的数据。基于这些，我们给出几种常见的用于评估候选分割的误差测度：

* 误分类率（Misclassification rate）.我们定义最有可能的标签为。则相应的误分类率为：

 （16.9）

* 熵（entropy）或者偏差（deviance）：

 （16.10）

值得注意的是，最小化熵等价于最大化测试*Xj* < *t*与类别标签*Y*之间的**信息增益**（information gain），定义为：

 （16.11）

 （16.12）

 （16.13）

而就是分布的MLE。

* 基尼系数（Gini index）

 （16.14）

上式是期望误差率。为了说明这一点，注意为在叶子节点中任意一个实例属于类别*c*的概率，而为该实例被误分类的概率。

在二分类问题中，，误分类率为，熵为H2(p)，基尼系数为2p(1-p)。图16.3给出了示例图。我们发现交叉熵和基尼系数十分相似，与误分类率相比，它们对类别概率的变化更加敏感。举例来说，考虑一个二分类问题，在每个类中有400个实例。假设一种分割导致的两个节点中数量分别为（300,100）和（100,300），然而，另一种分割导致的两个节点的数量分别为（200,400）和（200,0）。每一种分割所导致的误分类率都是0.25.然而，后者似乎更加好一些，因为其中一个节点是**纯**（pure）的，也就是它只包含一个类。而交叉熵和基尼系数会更加支持后面的选择。

16.2.2.3 例子

作为一个例子，考虑3类别鸢尾花数据集4个特征中的2个，如图16.4（a）所示。最终的决策树如图16.5（a）所示，决策边界如16.4（b）所示。我们发现这棵树与决策边界一样非常的复杂。在图16.5（b）中，我们发现通过交叉验证得到的错误率比训练集中的高，也就是说模型出现了过拟合。接下来我们将讨论如何进行树的剪枝，从而简化决策树。

16.2.3 树的剪枝

我们可以利用如下的方式防止过拟合的发生：如果一棵额外子树的生成所带来的误差率的减少不足以平衡因此而带来的复杂度的增加，那么我们就放弃生成额外的子树。然而这种说法过于模糊。举例来说，在图14.2（c）所示的亦或数据中，如果按照上述的准则，那么有可能不会产生任何的分割，因为每一个特征本身只具备很少的预测能力。

标准的方法是先生成一棵“完整”的树，然后进行修剪（pruning）。在具体操作中，我们可以剪去那些带来损失降低最小的分支。

为了确定回剪的深度，我们可以使用交叉验证的方法评估每一棵子树的误差，然后选择CV误差在最小标准误差1以内的树。图16.5（b）说明了这一点。含有最小CV误差的点对应着图16.6（a）中的简单的决策树。

16.2.4 树的优势和缺点

CART因为如下几个原因而有名：它们具备高解释性，可以很容易地处理混合的离散和连续输入，它们对输入的单调映射并不敏感（因为分割点是基于数据点的排序的），它们可以对变量进行自动的选择，且相对而言对异常点具备鲁棒性，它们对大规模数据具有好的扩展性，通过调整可以处理输入数据遗失的问题。

然而，CART模型也有一些缺点。最主要的一个就是相较于其他模型，它们的预测精度并不高。部分原因归咎于贪婪算法的本质。另一个相关的问题是树是不稳定（unstable）的，输入数据的很小的改变都可能对树的结构造成大的影响，原因在于树具有分层结构，导致高层次的误差会影响后面的部分。在频率学派的术语中，我们称树是一个高方差估计器。我们将在下面讨论一种解决方案。

16.2.5 随机森林

一种降低估计值方差的方法是对许多估计值进行平均化处理。举例来说，我们可以基于不同的数据子集（通过有放回的随机采样生成）训练M棵不同树，然后计算整个集合

 （16.15）

其中为第m棵树。这种技术被称为bagging（bootstrap aggregating）。

不幸的是，在数据的不同子集上运行同样的学习算法，可能会导致预测期的高相关性，从而限制了可能降低的方差的幅度。通过随机选择不同的特征以及数据样本，构造不同的子集，并基于这些子集去训练基学习器，从而实现基学习器的解耦，这个技术被称为随机森林（random forests）。这样的模型通常具有好的预测精度，并且广泛地应用在很多领域。

Bagging是一个频率学派的概念。我们也可以使用贝叶斯的方法去学习一棵树。特别地，通过MCMC方法对树的空间进行近似的推理。这将降低预测结果的方差。我们也可以基于集成树的空间进行贝叶斯推理，该方法的效果往往更好。这被称为贝叶斯自适应回归树（Bayesian adaptive regression trees, BART）。注意这些基于采样的贝叶斯方法的损失与基于采样的随机森林方法的损失不相上下。也就是说，每一种方法的训练都较慢，但其质量都较高。

不幸地是，使用复合树（无论是频率学派还是贝叶斯方法）失去了好的可解释性。幸运的是，可以使用不同的预处理方法，这一点将在16.8节介绍。

16.2.6

16.3 广义叠加模型

16.4 Boosting

Boosting是训练如式16.3形式的自适应基函数模型的贪心算法，其中为通过某个算法产生的**弱学习器**（weak learner）或者**基学习器**（base learner）。通过将弱学习器连续地应用在含权重的数据上，使得整个算法奏效，其中在训练的前期被分类错误的数据将被赋予更大的权重。

这个弱学习器可以是任意一种分类或者回归算法，但是常见的是CART模型。1998年，Leo Breiman称以浅层决策树为弱分类器的boosting为“best off-the-shelf classifier in the world”。这个观点被一些文献中的对比实验所支持，文献中对比了10种分类器，结果表明根据ROC曲线判断，无论在误分类错误方面还是在产生校准良好的概率方面，增强决策树都是最好的（排名第二的是随机森林，见16.2.5）。相反，单一的决策树性能很差。

Boosting最早出现在计算学习理论的文献中，其关注点主要是二分类。这些文献证明：如果弱学习器的性能总是比随机分类器的性能稍微高一点，那么我们可以（基于训练集）将一个弱学习器的性能提升到任意高的程度。举例来说，在图16.8中，我们绘制了增强决策树在训练集和测试集上的误差，所使用的数据为图16.10中所展示的数据。我们发现训练集上的误差很快下降到接近0的地步。更让人惊讶的是，尽管训练误差已经到达了0，但测试集上的误差继续在下降（尽管测试集上的误差最终还会上升）。所以提升算法是不容易过拟合的。（增强决策树是一个非常成功的人脸检测算法的基础，它被用于产生图1.6的结果，在很多数码相机中也被经常使用。）

基于它在经验上的惊人的成功，统计学家开始对这种方法产生兴趣。Breiman指出boosting可以理解为一种在函数空间上的梯度下降。这个观点在某些论文中被拓展，该论文展示了如何将boosting拓展到不同的损失函数中，包括回归，健壮回归，泊松回归等等。本节，我们将呈现关于boosting的统计学方面的解释。

16.4.1 前向逐步叠加建模

Boosting的目的是求解如下的优化问题：

 （16.25）

其中为某一种损失函数，*f*为如式16.3所示的ABM模型。常见的损失函数见表16.1所示。

如果我们使用平方差损失，最优的估计函数为：

 （16.26）

这一点在5.7.13节给出说明。当然，这在实际过程中的并不能计算，因为它需要知道真实的条件分布*p*(y|**x**)。所以有时这也被称为全局最小估计函数（population minimizer），其中期望属于频率学派的解释。接下来，我们将会看到boosting方法将尝试逼近这个条件期望。

对于二值分类问题，常用的损失函数为0-1损失，但它不是可微的函数。取而代之的是，我们通常使用对数损失，它是0-1损失的凸上限，我们在6.5.5节说明过这一点。在这种情况下，其最优的预测函数为：

 （16.27）

其中。我们可以将这种框架拓展到多分类中，但此处我们不做介绍。

另一种可选的凸上界为指数损失（exponential loss），定义为：

 （16.28）

图16.9给出了图形表示。该损失函数相较于对数损失函数在计算上更具优势，这一点将在后面讨论。结果表明，针对该损失函数的最优估计器也是。为了说明这一点，我们将（每个x）损失的期望的导数设置为0，得到：

 （16.29）

 （16.30）

 （16.31）

所以，在两种情况下，我们可以发现boosting尝试估计对数几率（的1/2）。

既然找到最优的*f*很困难，那么我们可以采用序列化地方式不断地求解。我们定义初始化函数：

 （16.32）

举例来说，如果我们使用平方差损失函数，我们可以设置，如果我们使用对数损失或者指数损失，我们可以设置，其中。我们也可以使用一个更加有力的模型作为我们的基准，比如GLM。

然后在第m次迭代，我们计算：

 （16.33）

然后令：

 （16.34）

训练的关键点在于我们并不会回溯之前的过程去调节之前的参数。这也是为什么这种方法称为**前向逐步叠加建模**（forward stagewise additive modeling）。

我们对上述过程迭代M次。事实上，M是这种方法中主要需要调节的超参。通常情况下，我们使用一个验证集去监督模型的性能，当在验证集上的性能开始下降时，我们就终止训练，这种方法被称为**提前终止**（early stopping）。另一种方案是，我们可以使用模型选择标准，比如AIC或者BIC。

在实际过程中，通过采用“部分更新”的形式，可以获得更好地（测试集）性能

 （16.34）

其中被称为步长参数。实践中经常使用，这被称为**收缩**（shrinkage）。

接下来我们将讨论如何解决式16.33的子问题。这与损失函数的形式有关，与弱学习器的类型无关。

16.4.2 L2boosting

假设我们使用平方差损失函数。然后在第*m*步损失函数的形式为：

 （16.36）

其中为当前阶段的残差，不失一般性，我们令。所以通过使用弱分类器来预测*r*m，我们可以找到一个新的基函数。这被称为L2boosting或者least squares boosting。在16.4.6节，我们将会发现，通过选择一个合适的弱学习器，这种方法可以给出跟LARS一样的结果，后者可以进行变量选择（章节13.4.2）.

16.4.3 AdaBoost

考虑一个使用指数损失的二分类问题。在第m步，我们需要最小化

 （16.37）

其中为添加到数据i上的权重，。我们可以重写这个目标函数为：

 （16.38）

 （16.39）

因此需要被添加的最优的函数为：

 （16.40）

通过将弱分类器应用到含权重的数据集上，我们可以找到这些函数。将代入到*L*m，我们求得值为：

 （16.41）

其中

 （16.42）

那全局的更新为：

 （16.43）

基于此，下一次迭代中的权重将变成：

 （16.44）

 （16.45）

 （16.46）

其中我们使用了如下的事实：如果，则，否则。因为在归一化的过程中会被约掉，所以我们将它省略。结果得到算法16.2，被称为Adaboost.M1.

该算法在实际应用中的一个例子如图16.10所示，其中使用了决策树作为弱学习器。我们发现在许多次迭代之后，我们可以得到一个复杂的决策边界。更让人惊讶的是，AdaBoost过拟合的速度非常慢，如图16.8所示。16.4.8对这一点进行了讨论。

|  |
| --- |
| 算法16.2：Adaboost.M1，指数损失下的二值分类问题 |
| 1.  2. for m=1:M do  3. 基于权重w训练一个分类器;  4. 计算;  5. 计算  6. 令  7. 返回 |

16.4.4 LogitBoost

指数损失的问题在于，它给错误分类的示例增加了很多权重，从图16.9左边的指数放大可以明显看出这一点。这就导致方法对异常值（被错误标记的案例）十分敏感。除此之外，对于二值变量而言，不是任意概率质量函数的对数。因此我们无法从函数f(x)中恢复概率估计。

一种自然的替代方案是使用对数损失。这种方式只会线性地惩罚那些错误，图16.9给出了说明。更进一步，这意味着我们可以从最后学习到的函数提取出概率值，使用：

 （16.47）

其目标是最小化对数损失的期望值，定义为：

 （16.48）

通过使用牛顿法（类似于IRLS），我们可以推导出算法16.3。这被称为logitBoost。它可以泛化到多类别的情况中。

|  |
| --- |
| 算法16.3：LogitBoost，基于log-loss的二分类问题 |
| 1.  2. for m=1:M do  3. 计算工作响应  4. 计算权重  5.  6. 更新  7. 计算  8.返回 |

16.4.5 作为函数梯度下降的提升

为了避免针对不同的损失函数推导新的版本的boosting，存在一种推导出通用版本的可能，被称为**梯度提升**（gradient boosting）。为了解释这一点，假设我们需要最小化：

 （16.49）

其中为“参数”。我们使用梯度下降的方法逐步地解决上述问题。在第m步，令*g*m为*L*(f)在f=fm-1处的梯度：

 （16.50）

一些常用的损失的梯度如表16.1所示。接着我们对函数进行更新：

 （16.51）

其中为步长，由下式决定：

 （16.52）

这被称为**泛函梯度下降**（functional gradient descent）。

使用当前的形式并没有什么用，因为它只是在固定的*N*个点对函数**f**优化，所以我们并没有学习到一个能够泛化的函数。然而，我们可以训练一个弱分类器来近似负梯度信号，从而实现对算法的调整。也就是说我们使用下式进行更新：

 （16.53）

整体的算法见算法16.4.（我们已经省略了线性搜索的步骤，它并不是必须的步骤）。

|  |
| --- |
| 算法16.4：梯度提升 |
| 1. 初始化  2. **for** *m*=1:*M* **do**  3. 计算残差梯度；  4. 使用弱学习器计算，从而最小化  5. 更新  6. 返回 |

如果基于平方差损失使用该算法，我们将得到L2Boosting。如果我们应用到对数损失，将得到BinomialBoost。该算法优于LogitBoost的原因在于它不需要能够实现含权重的训练：它只需要将任意的黑箱回归模型应用到梯度向量。同时，将其拓展到多分类问题中也相对容易。我们可以将该算法应用到其他损失函数，比如Huber loss，它比平方差损失对异常值的鲁棒性更高。

16.4.6 稀疏boosting

16.4.7 多变量自适应回归树

将CART模型作为弱学习器是非常常见的。通常建议使用一个比较浅的树，这样方差会比较小。尽管偏差会比较大（因为一棵浅的树似乎离“真相”更远），但这会在后面的提升过程中得到弥补。

树的高度是一个额外的需要调节的参数（除了M，提升的次数；，收缩的因子）。假设我们限制树的叶子数为*J*。如果*J*=2，我们将得到一个分叉树，这种情况下我们只能基于一个变量进行分割。如果*J*=3，我们将允许2个变量进行分割，以此类推。通常情况下，建议使用*J*≈6。

如果我们将梯度提升算法与（浅）回归树相组合，我们将得到一个被称为MART的模型，即multivariate adaptive regression trees。实际上这对基本的梯度提升算法进行了稍微的强化：在基于残差（负梯度）对回归树进行训练之后，我们在树的叶子节点对参数进行重新估计，以最小化如下的损失：

 （16.58）

其中表示第m棵树的第*j*个叶子。为对应的参数（在回归问题中为*y*的平均响应，或者在分类问题中，表示最有可能的类别标签）。

16.6 集成学习

集成学习是指学习一个基模型的加权组合，形式为：

（16.102）

其中为可调节的参数。集成学习有时又被称为**委员会方法**（committee method），因为每一个基模型fm都获得一个加权“投票”。

显然，集成学习与学习自适应基函数模型有精密联系。事实上，我们可以将一个神经网络当作一种集成方法，其中fm代表第m个隐节点，wm为输出层的权重值。同时，我们可以将提升算法当作一种集成学习，其中基模型的权重是通过连续的方式确定的。接下来我们描述几种集成学习的形式。

16.6.1 Stacking

一种估计式16.102中的权重的方式是使用：

（16.103）

然而，这样会导致过拟合，使得过大，模型变得复杂。一种简单的解决方案是使用交叉验证。特别地，我们可以使用LOOCV估计

（16.104）

其中为基于除了（xi,yi）之外的数据集训练得到的预测器。这被称为stacking，即所谓的“stacked generalization”。这种方法比标准的