Tree based methods

Machine Learning

Prof. Neylson Crepalde

Some of the figures in this presentation are taken from "An Introduction to Statistical Learning, with applications in R" (Springer, 2013) with permission from the authors: G. James, D. Witten, T. Hastie and R. Tibshirani

Decision Trees

Os modelos baseados em árvore consistem em **estratificar** ou **segmentar** o espaço do preditor em um número de regiões simples. Para fazer predições para uma observação, normalmente usamos a média ou a moda da região a qual a observação pertence. Como as regras de separação usadas para segmentar o espaço do preditor podem ser resumidas em uma árvore, essas abordagens são conhecidas como *decision trees*.

A *decision tree* pode ser aplicada tanto a problemas de regressão quanto a problemas de classificação.

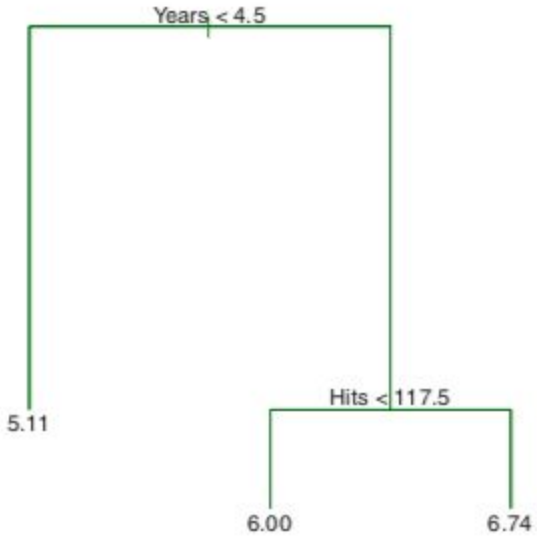

FIGURE 8.1. *For the* `Hitters` *data, a regression tree for predicting the log salary of a baseball player, based on the number of years that he has played in the major leagues and the number of hits that he made in the previous year. At a given internal node, the label (of the form* $X_j < t_k$*) indicates the left-hand branch emanating from that split, and the right-hand branch corresponds to* $X_j \geq t_k$*. For instance, the split at the top of the tree results in two large branches. The left-hand branch corresponds to* `Years<4.5`*, and the right-hand branch corresponds to* `Years>=4.5`*. The tree has two internal nodes and three terminal nodes, or leaves. The number in each leaf is the mean of the response for the observations that fall there.*

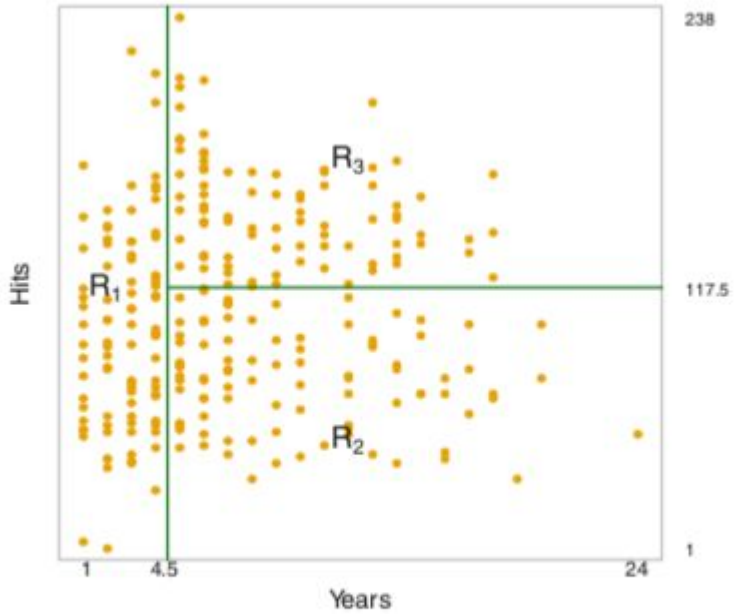

FIGURE 8.2. *The three-region partition for the* `Hitters` *data set from the regression tree illustrated in Figure 8.1.*

O algoritmo divide o espaço de preditores em três regiões que podem ser descritas assim:

R1 = {X | Years < 4.5}
R2 = {X | Years >= 4.5, Hits < 117.5}
R3 = {X | Years >= 4.5, Hits >= 117.5}

R1, R2 e R3 são chamados de *terminal nodes* ou *leaves (folhas)* de uma árvore. Os pontos onde há separação de regiões são chamados *internal nodes.*

Nos referimos aos segmentos das árvores que conectam os nodes como *branches*.

Ciências de Dados

Podemos interpretar a árvore apresentada da seguinte forma: *Years* é a variável mais importante para prever o Salário e jogadores com menos experiência tem salários menores do que jogadores mais experientes. Dado que um jogador é menos experiente, o número de *Hits* que ele conseguiu na temporada anterior não tem muito impacto no salário. Mas entre os jogadores mais experientes, o número de *Hits* feitos no ano anterior afeta o salário de modo que jogadores com mais *Hits* no ano anterior tendem a ter salários maiores.

Obviamente, essa explicação é uma ultra-simplificação da relação real entre *Hits, Years* e *Salary*. Entretanto, além de possuir uma boa visualização, as regressões baseadas em árvore são bastante fáceis de interpretar.

Como as árvores são construídas?

O processo possui basicamente dois passos:

1) O espaço preditor é dividido em *J* regiões distintas e sem sobreposição R1, R2, … R*j*.
2) Para cada observação que cai na região R*j* fazemos a mesma predição, a média dos valores para os dados de treino na região R*j.*

Como as divisões de regiões são feitas? Dividimos o espaço preditor em retângulos multidimensionais, ou *boxes*, que minimizem o RSS.

$$\sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2$$

Infelizmente é computacionalmente inviável considerar todas as possíveis partições do espaço preditor em *J* regiões. Portanto, realizamos uma abordagem *top-down* conhecida como *recursive binary splitting*. Começamos no topo da árvore e fazemos *splits* gerando 2 *branches*. Desse modo, o melhor *split* é realizado localmente.

Para realizar a *recursive binary splitting* primeiro selecionamos uma variável X e um *cutpoint s* que divida o espaço preditor em duas regiões com o menor RSS. São considerados todos os X e todos os possíveis *cutpoints* para cada preditor. Formalmente, para cada *j* e *s* , definimos pares de regiões

$$R_1(j,s) = \{X | X_j < s\} \text{ and } R_2(j,s) = \{X | X_j \geq s\}$$

de modo a minimizar a equação

$$\sum_{i:\, x_i \in R_1(j,s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i:\, x_i \in R_2(j,s)} (y_i - \hat{y}_{R_2})^2$$

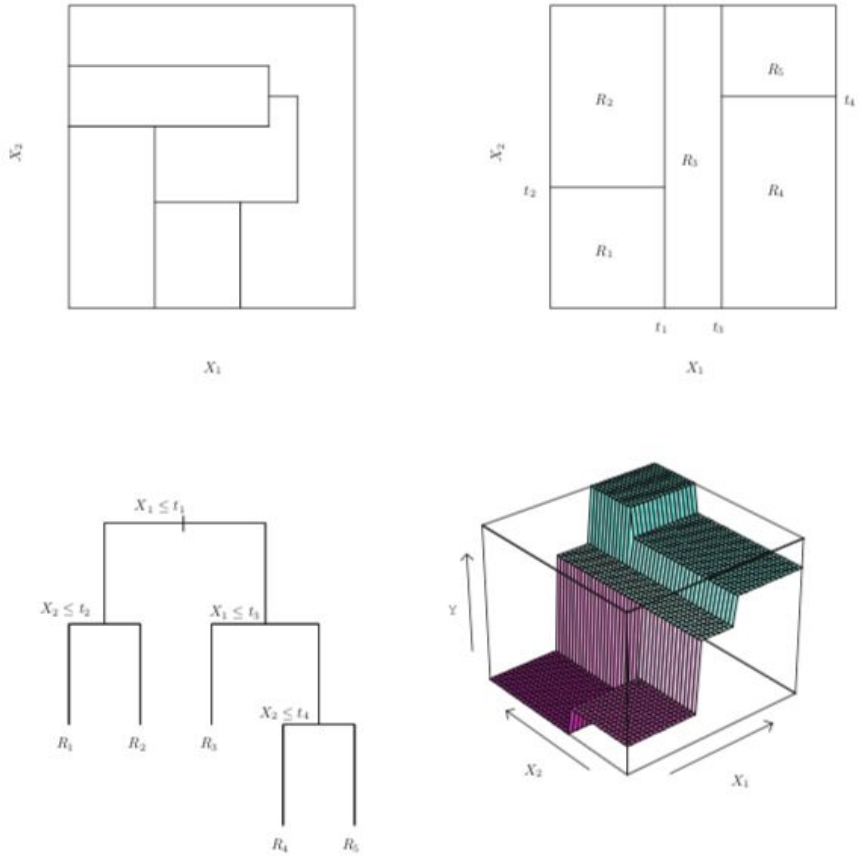

FIGURE 8.3. Top Left: *A partition of two-dimensional feature space that could not result from recursive binary splitting.* Top Right: *The output of recursive binary splitting on a two-dimensional example.* Bottom Left: *A tree corresponding to the partition in the top right panel.* Bottom Right: *A perspective plot of the prediction surface corresponding to that tree.*

Tree Prunning (Poda)

O processo descrito pode produzir boas estimativas em dados de treino mas muito provavelmente vai resultar em *overfitting* dado que algumas árvores produzidas podem ficar muito complexas. Uma árvore menor com menos *splits* e menos regiões pode ter menor variância menor e melhor interpretação ao custo de um pequeno viés. Uma boa estratégia para realizar essa tarefa é produzir uma árvore bastante grande *T0* e *podá-la (prune it)* para obter uma sub-árvore. Intuitivamente, nosso objetivo é selecionar uma sub-árvore que tenha o menor erro nos dados de teste/validação. Podemos utilizar a técnica do set de validação ou realizar cross-validation. Entretanto, realizar cross-validation para todas as *subtrees* possíveis pode ser extremamente custoso. Ao invés disso, precisamos de um método para selecionar uma árvore podada.

O método *Cost complexity prunning,* também conhecido como *weakest link prunning,* ao invés de considerar todas as subtrees, considera uma sequência de subtrees indexadas por um tunning parameter não negativo *alpha.* Para cada valor de *alpha* corresponde uma subtree contida em *T0* tal que

$$\sum_{m=1}^{|T|} \sum_{i:\, x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha |T|$$

seja o menor possível. T indica o número de *terminal nodes* ou *leaves, Rm* é o retângulo correspondente ao *m-ésimo terminal node,* e $\hat{y}_{R_m}$ a resposta predita associada a *Rm*, isto é, a média das observações na região *Rm*. O parâmetro *alpha* controla um trade-off entre a complexidade da rede e o seu ajuste. Quando $\alpha = 0$ a subtree *T* será igual a *T0*. Quando *alpha* tende ao infinito, há um preço pela quantidade de folhas portanto quantidade de terminal nodes diminui. A equação acima tem uma relação próxima com o Lasso.

Algorithm 8.1 *Building a Regression Tree*

1. Use recursive binary splitting to grow a large tree on the training data, stopping only when each terminal node has fewer than some minimum number of observations.

2. Apply cost complexity pruning to the large tree in order to obtain a sequence of best subtrees, as a function of α.

3. Use K-fold cross-validation to choose α. That is, divide the training observations into K folds. For each $k = 1, \ldots, K$:

   (a) Repeat Steps 1 and 2 on all but the kth fold of the training data.

   (b) Evaluate the mean squared prediction error on the data in the left-out kth fold, as a function of α.

   Average the results for each value of α, and pick α to minimize the average error.

4. Return the subtree from Step 2 that corresponds to the chosen value of α.

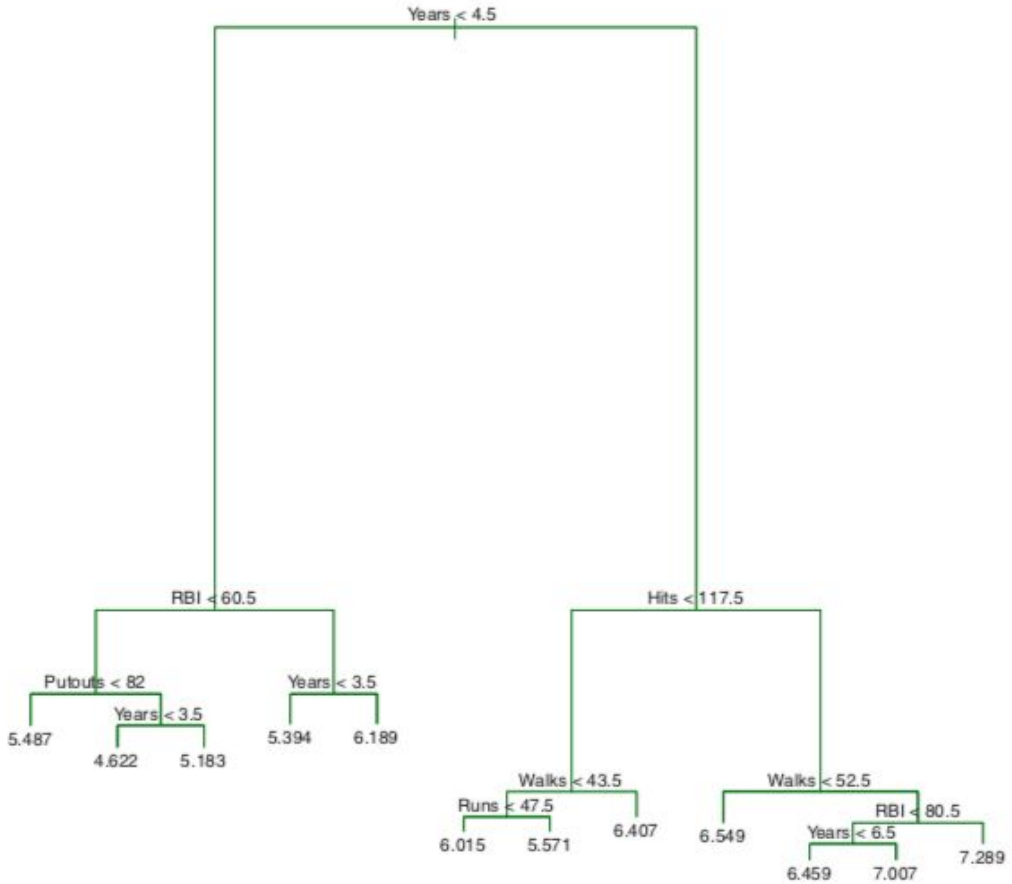

FIGURE 8.4. *Regression tree analysis for the* `Hitters` *data. The unpruned tree that results from top-down greedy splitting on the training data is shown.*

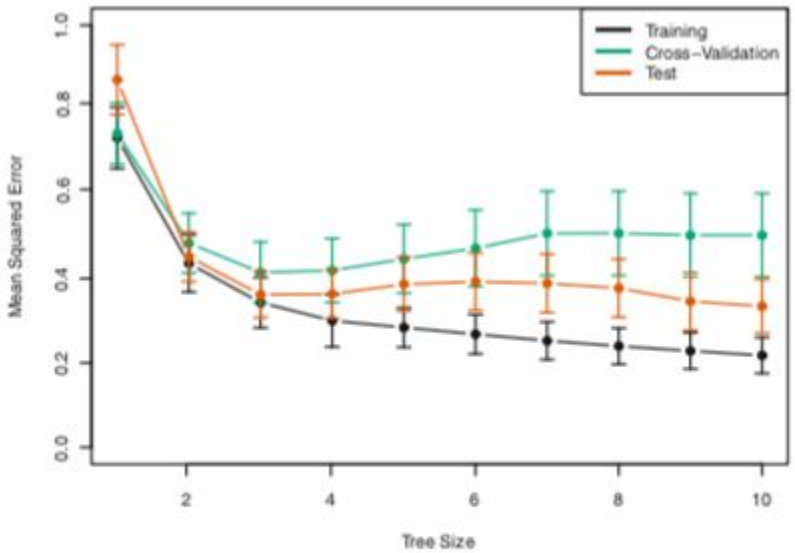

FIGURE 8.5. *Regression tree analysis for the* `Hitters` *data. The training, cross-validation, and test MSE are shown as a function of the number of terminal nodes in the pruned tree. Standard error bands are displayed. The minimum cross-validation error occurs at a tree size of three.*

Classification trees

Árvores de classificação funcionam de maneira muito similar às árvores de regressão. A diferença é que as predições são feitas com o *valor mais comum* (moda). Ao interpretar os resultados estamos interessados em prever a classe correspondente a uma determinada região mas também na proporção de classes que caem naquela região.

O processo de construção da árvore é exatamente igual ao das *regression trees.* Aqui, entretanto, ao invés de RSS podemos usar *classification error rate*. A *classification error rate* é simplesmente a fração das observações na região que não pertencem à classe mais comum:

$$E = 1 - \max_k(\hat{p}_{mk})$$

Contudo, essa medida de erro não é sensitiva o suficiente para as árvores. Outras duas medidas são preferíveis.

O **Índice de Gini** pode ser definido por:

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}).$$

Ele é uma medida da variância total entre as K classes. O Gini toma um valor pequeno quando todas as proporções *Pmk* são próximas de 0 ou 1. Por esse motivo, o índice de Gini é conhecido como uma medida de *pureza* dos nós. Um Gini pequeno indica que um *node* possui predominantemente observações de uma única classe.

Uma alternativa ao Gini é a medida de **entropia**, dada por

$$D = -\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}.$$

Visto que $0 \leq \hat{p}_{mk} \leq 1$, concluímos que $0 \leq -\hat{p}_{mk} \log \hat{p}_{mk}$ A **entropia** vai assumir um valor próximo de 0 quando se todas as proporções *Pmk* são próximas de 0 ou 1.

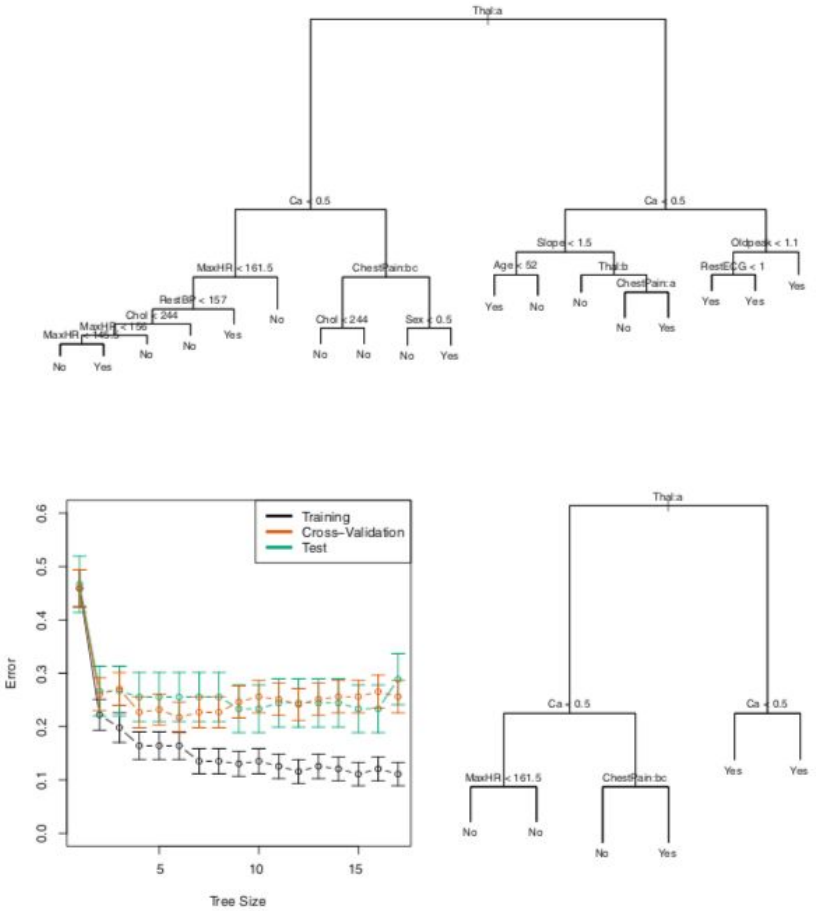

FIGURE 8.6. `Heart` *data.* Top: *The unpruned tree.* Bottom Left: *Cross -validation error, training, and test error, for different sizes of the pruned tree.* Bottom Right: *The pruned tree corresponding to the minimal cross-validation error.*

Trees vs Linear Models

Qual método será melhor?
Depende do caso!

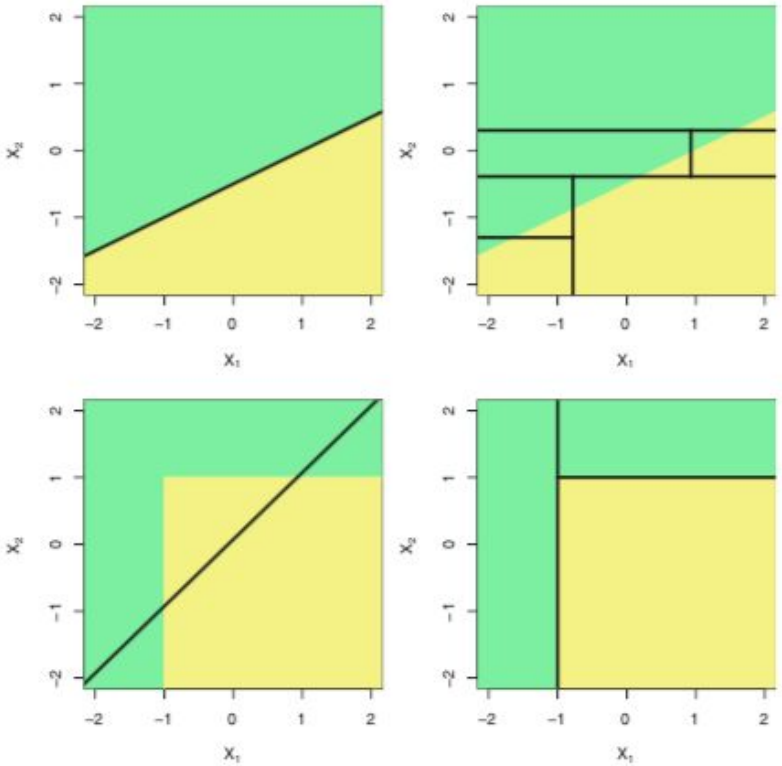

FIGURE 8.7. Top Row: *A two-dimensional classification example in which the true decision boundary is linear, and is indicated by the shaded regions. A classical approach that assumes a linear boundary (left) will outperform a decision tree that performs splits parallel to the axes (right).* Bottom Row: *Here the true decision boundary is non-linear. Here a linear model is unable to capture the true decision boundary (left), whereas a decision tree is successful (right).*

Bagging e Random Forests

As *decision trees* possuem alta variância. Podemos usar o procedimento de *Bootstrap aggregation* ou *bagging* para reduzir a variância do método de aprendizado estatístico. Após gerar *B* novas amostras por *bootstrap*, fazemos a predição para a *b-ésima* amostra gerada e calculamos a média para todas as amostras da seguinte forma:

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{*b}(x)$$

Isto é o *bagging.* Não executamos o *prunning* mas, ao invés disso, combinamos várias árvores com alta variância e obtemos um modelo ao final com variância reduzida pela média das predições. O número de árvores *B* estimadas não é um valor crítico no *bagging*. Usar um valor alto de *B* não vai causar *overfitting.* Na verdade, podemos escolher um valor de *B* suficientemente grande para obter um bom ajuste.

Ao realizar predições com *bagging*, perdemos a capacidade de interpretar os resultados a partir de uma única árvore pois os resultados são a média de centenas ou, às vezes, milhares de árvores. *Bagging,* portanto, melhora muito a acurácia da estimação ao custo da interpretabilidade.

Embora não tenhamos uma interpretação clara, podemos extrair um resumo geral da **importância das variáveis** através de RSS no caso de problemas de regressão e do índice de Gini quando forem problemas de classificação.

Out-of-bag error estimations

Há um modo muito prático para estimar o erro num modelo de *bagging* sem a necessidade de fazer cross-validation. Ao fazer *bootstrap* nos dados de treino, é possível demonstrar que em torno de 2 terços dos dados são usados. O outro terço que sobra é chamado de observações *out-of-bag* (fora da mochila/saco). Podemos então predizer uma resposta para a *i-ésima* observação usando cada uma das árvores para as quais essa observação foi OOB. Isso dá B/3 predições para a observação *i.* Para obter uma única predição podemos tirar a média das predições (no caso de regressão) ou contar a categoria mais votada (no caso de classificação).

É possível mostrar que quando *B* é suficientemente grande, OOB tem resultados similares ao LOOCV sem o custo computacional.

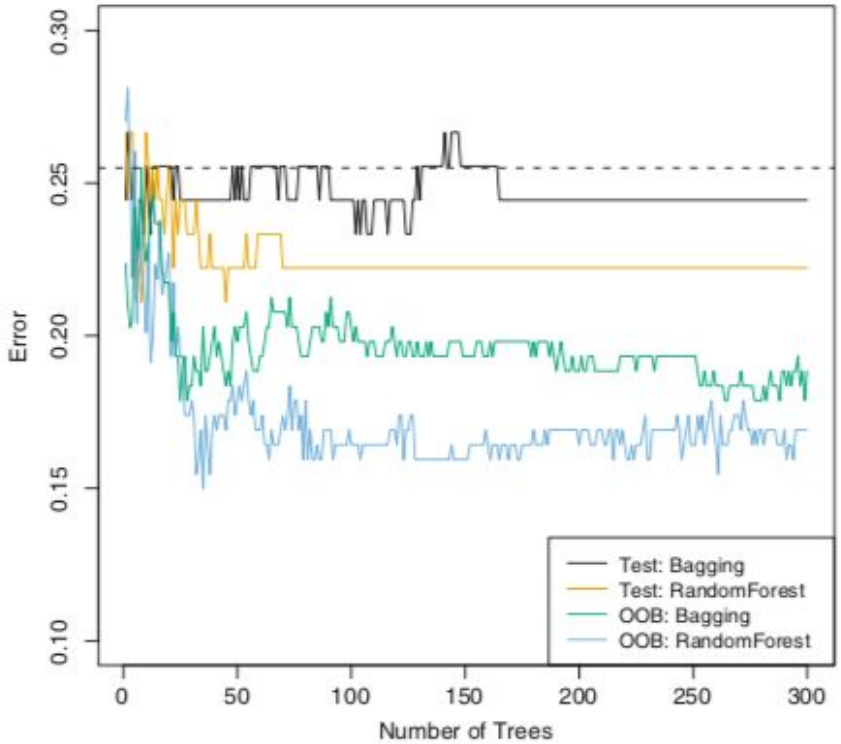

FIGURE 8.8. *Bagging and random forest results for the* `Heart` *data. The test error (black and orange) is shown as a function of* B*, the number of bootstrapped training sets used. Random forests were applied with* $m = \sqrt{p}$*. The dashed line indicates the test error resulting from a single classification tree. The green and blue traces show the OOB error, which in this case is considerably lower.*

As ***Random Forests*** apresentam uma sofisticação no *bagging* introduzindo um modo de descorrelacionar as árvores. Como no *bagging*, construímos várias *decision trees* em um banco de treino *boostrapped*. Contudo, ao construir as árvores, em cada *split* (divisão) uma *amostra aleatória de m preditores* é considerada para fazer o *split*. No *split* só é permitido usar um dos *m* preditores. A cada novo *split* uma nova amostra aleatória de *m* preditores é selecionada. Normalmente são usados $m \approx \sqrt{p}$ possibilidades para *split*.

Como em *bagging*, um alto valor de *B* não leva a *overfitting*. Ao contrário, podemos escolher um valor de *B* alto o suficiente para obter um bom ajuste.

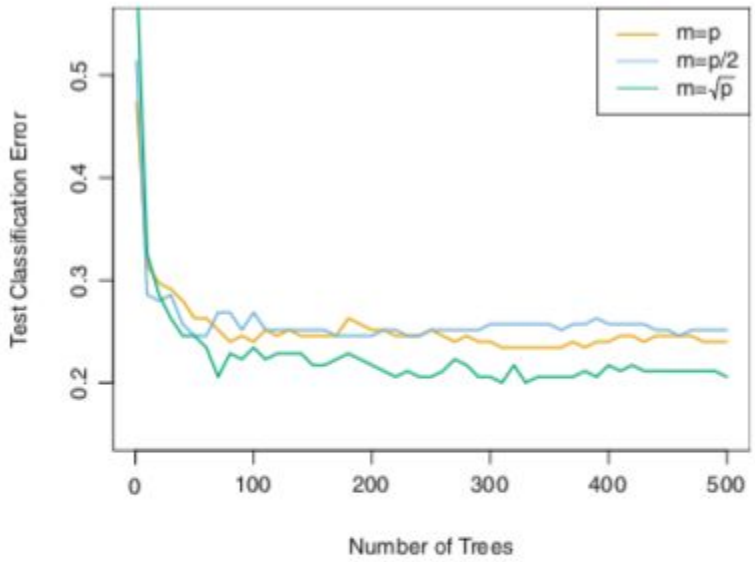

FIGURE 8.10. *Results from random forests for the 15-class gene expression data set with $p = 500$ predictors. The test error is displayed as a function of the number of trees. Each colored line corresponds to a different value of m, the number of predictors available for splitting at each interior tree node. Random forests ($m < p$) lead to a slight improvement over bagging ($m = p$). A single classification tree has an error rate of 45.7%.*

Boosting

Boosting é, na verdade, um procedimento estatístico que pode ser aplicado a diversos métodos de regressão ou classificação. Aqui, discutimos *boosting* apenas no contexto das *decision trees.*

Boosting funciona de maneira parecida com o *bagging* gerando várias árvores. A diferença é que *bagging* combina todas as árvores geradas. Já *boosting* estima as árvores de decisão ***sequencialmente***, isto é, cada árvore é estimada usando informação da árvore anterior. *Boosting*, portanto, **não faz *bootstrap*** mas **estima cada árvore numa versão modificada do dataset original.**

No *boosting,* a partir do modelo anterior, estimamos uma árvore para os **resíduos** desse modelo ao invés de Y. Depois, adicionamos essa nova árvore às funções estimadas para atualizar os resíduos. Cada árvore pode ser bem pequena, com apenas algumas folhas, o que é determinado pelo parâmetro ***d*** do algoritmo. Desse modo, melhoramos cada vez mais a função estimada em áreas onde ela não tem boa performance. Em geral, esse tipo de abordagem estatística (*slow learning*) tende a ter bons resultados.

Algorithm 8.2 *Boosting for Regression Trees*

1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all i in the training set.

2. For $b = 1, 2, \ldots, B$, repeat:

   (a) Fit a tree $\hat{f}^b$ with d splits ($d+1$ terminal nodes) to the training data (X, r).

   (b) Update $\hat{f}$ by adding in a shrunken version of the new tree:

   $$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x). \tag{8.10}$$

   (c) Update the residuals,

   $$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i). \tag{8.11}$$

3. Output the boosted model,

$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x). \tag{8.12}$$

Boosting possui 3 *tunning parameters:*

1. O número de árvores ***B***. Diferente de *bagging* e *Random Forests,* aqui se *B* for muito alto pode haver *overfitting*. Usamos cross-validation para escolher *B.*

2. O parâmetro de regularização ***lambda*****,** um número positivo e pequeno que controla a taxa de aprendizado de *boosting*. Valores comuns são 0.01 ou 0.001 e a boa escolha depende do problema. *Lambda* muito pequeno pode requerer um valor muito alto de *B* para obter boa performance.

3. O número de *splits* ***d*** em cada árvore que controla a complexidade do *boosted ensemble*. Normalmente *d* = 1 funciona bem. Nesse caso o boosted ensemble se torna um modelo aditivo de modo que cada termo envolve uma única variável. Esse parâmetro também é chamado de *interaction depth*.