Machine Learning, Neural Networks and Deep Learning

Dr. B. Wilbertz[1]

[1]Trendiction S.A. / Talkwalker

8th October, 2021

Tree-based Methods

Tree-based Methods

- Here we describe *tree-based* methods for regression and classification.
- These involve *stratifying* or *segmenting* the predictor space into a number of simple regions.
- Since the set of splitting rules used to segment the predictor space can be summarized in a tree, these types of approaches are known as *decision-tree* methods.

Pros and Cons

Pros and Cons

- Vanilla Tree-based methods are simple and useful for interpretation.
- However they typically are not competitive with the best supervised learning approaches in terms of prediction accuracy.
- Hence we also discuss *bagging*, *random forests*, and *boosting*. These methods grow multiple trees which are then combined to yield a single consensus prediction.
- Combining a large number of trees can often result in dramatic improvements in prediction accuracy, at the expense of some loss interpretation.

The Basics of Decision Trees

The Basics of Decision Trees

- Decision trees can be applied to both regression and classification problems.
- We first consider regression problems, and then move on to classification.

Baseball salary data (`hitter.csv`)

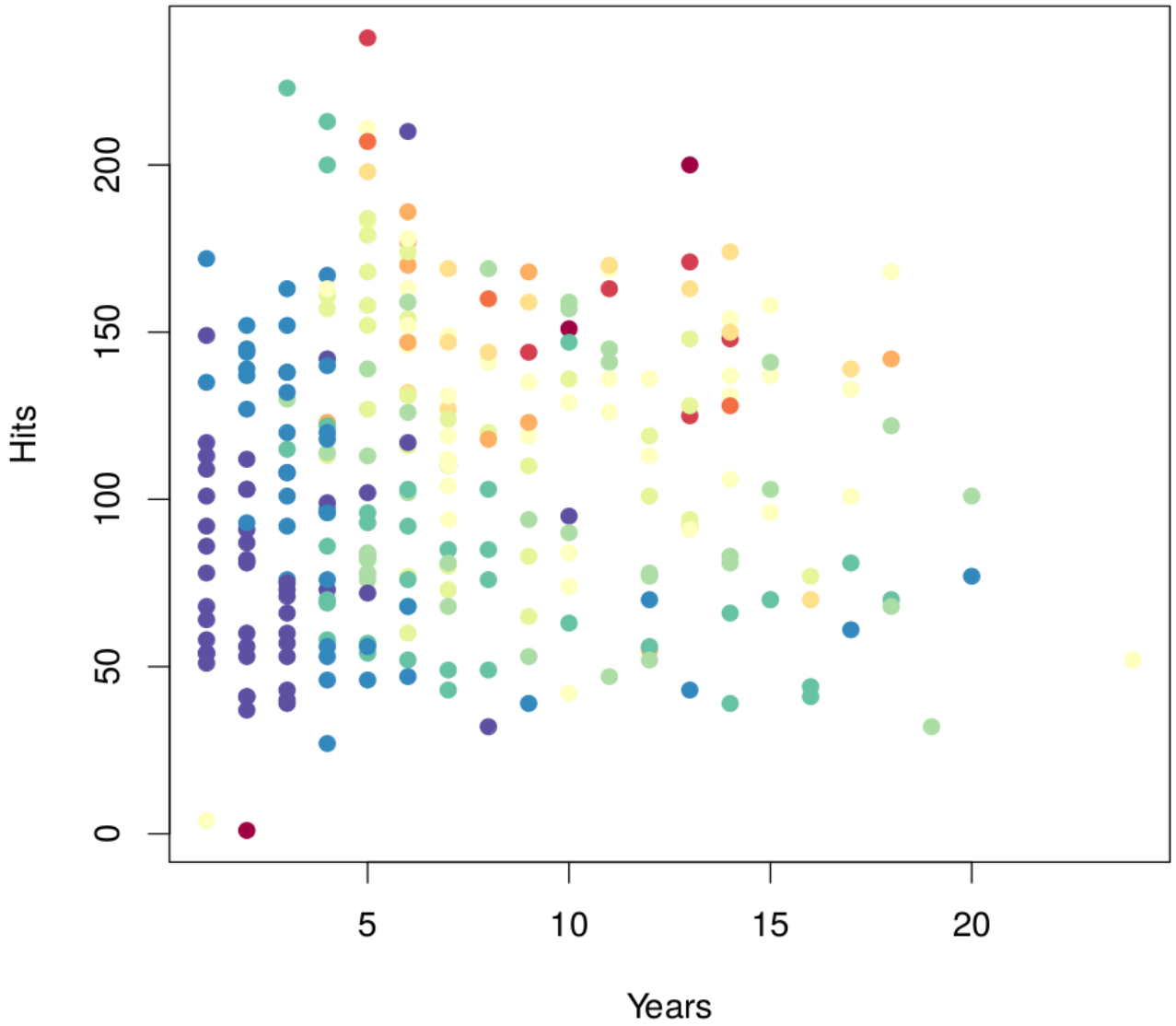
Example

Major League Baseball Data from the 1986 and 1987 seasons containing 320 observations and 20 variables. (Variables starting with `C` refer to the whole career)

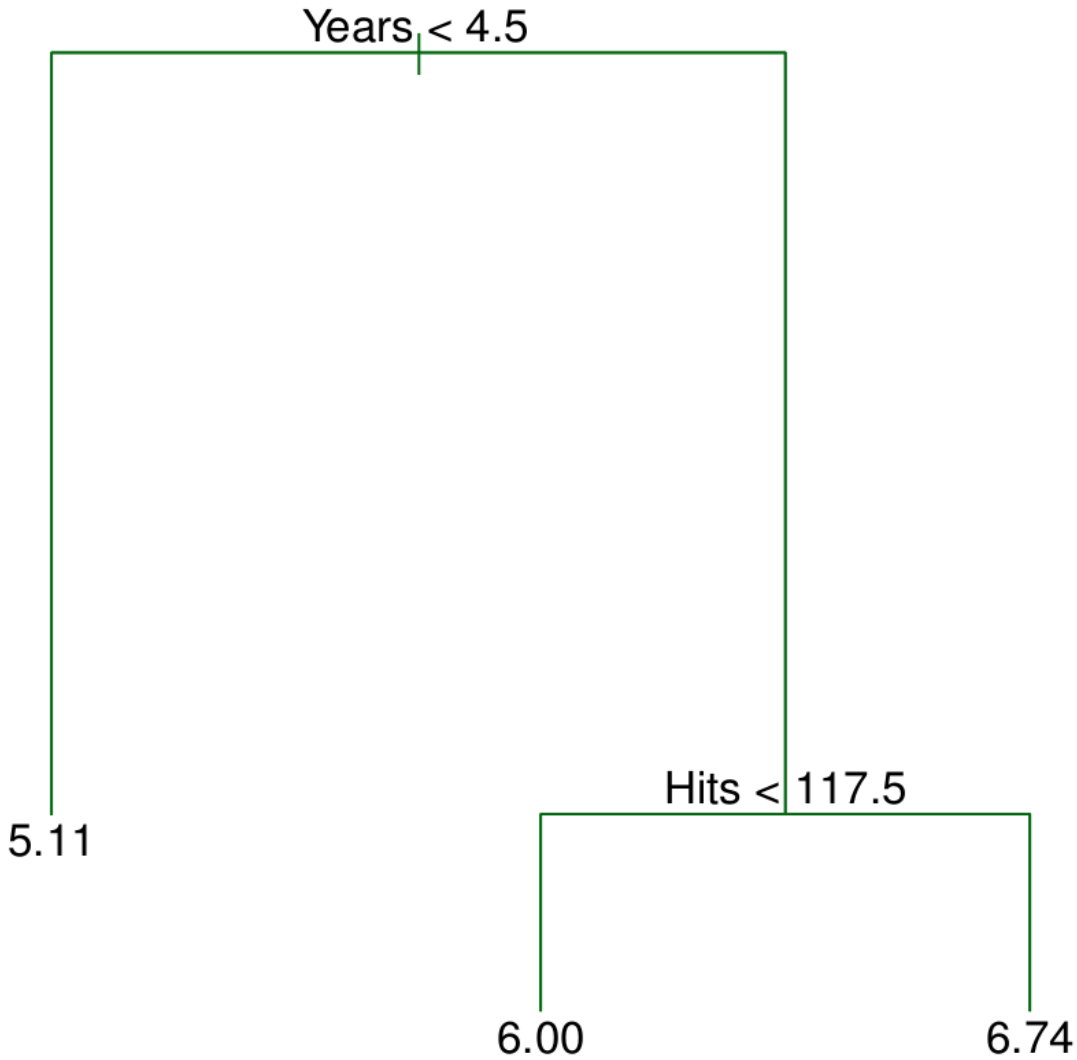
- `AtBat` Number of times at bat in 1986
- `Hits` Number of hits in 1986
- `HmRun` Number of home runs in 1986
- `Runs` Number of runs in 1986
- `RBI` Number of runs batted in in 1986
- `Walks` Number of walks in 1986
- `Years` Number of years in the major leagues
- `League` A factor with levels A and N indicating player's league at the end of 1986
- `Division` A factor with levels E and W indicating player's division at the end of 1986
- `PutOuts` Number of put outs in 1986
- `Assists` Number of assists in 1986
- `Errors` Number of errors in 1986
- `Salary` 1987 annual salary on opening day in thousands of dollars
- `NewLeague` A factor with levels A and N indicating player's league at the beginning of 1987

Baseball salary data: how would you stratify it?

Salary is color-coded from low (blue, green) to high (yellow,red)

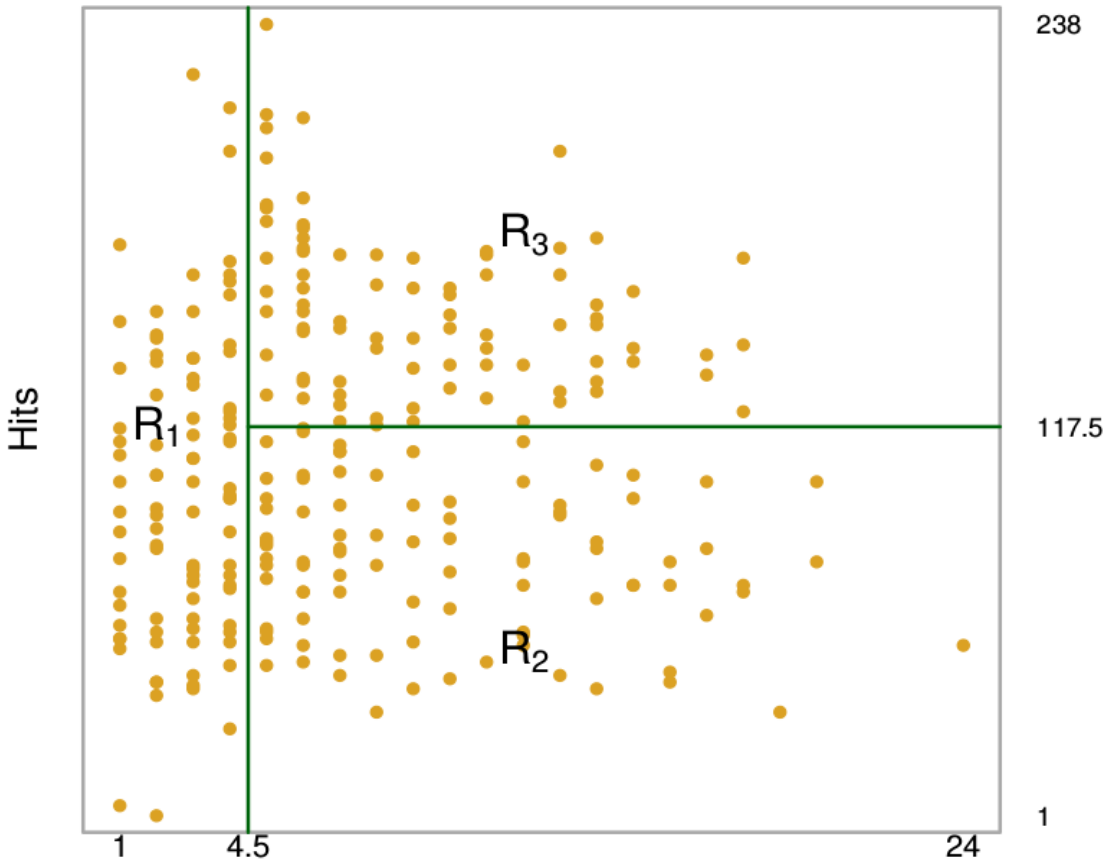


Decision tree for these data



Results

- Overall, the tree stratifies or segments the players into three regions of predictor space:
  $R_1 = \{X | \texttt{Years} < 4.5\}$, $R_2 = \{X | \texttt{Years} \geq 4.5, \texttt{Hits} < 117.5\}$, and $R_3 = \{X | \texttt{Years} \geq 4.5, \texttt{Hits} \geq 117.5\}$.



Details of the tree-building process

Details of the tree-building process

1. We divide the predictor space – that is, the set of possible values for $X_1, X_2, \ldots, X_p$ – into J distinct and non-overlapping regions, $R_1, R_2, \ldots, R_J$.
2. For every observation that falls into the region R_j, we make the same prediction, which is simply the mean of the response values for the training observations in R_j.

More details of the tree-building process

More details of the tree-building process

- In theory, the regions could have any shape. However, we choose to divide the predictor space into high-dimensional rectangles, or *boxes*, for simplicity and for ease of interpretation of the resulting predictive model.
- The goal is to find boxes $R_1, \ldots, R_J$ that minimize the RSS, given by

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

where $\hat{y}_{R_j}$ is the mean response for the training observations within the jth box.

More details of the tree-building process

- Unfortunately, it is computationally infeasible to consider every possible partition of the feature space into J boxes.
- For this reason, we take a *top-down, greedy* approach that is known as recursive binary splitting.
- The approach is *top-down* because it begins at the top of the tree and then successively splits the predictor space; each split is indicated via two new branches further down on the tree.
- It is *greedy* because at each step of the tree-building process, the *best* split is made at that particular step, rather than looking ahead and picking a split that will lead to a better tree in some future step.

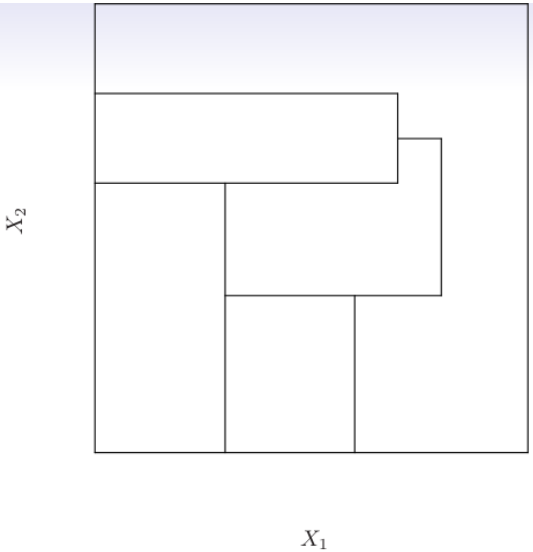
Details – Continued

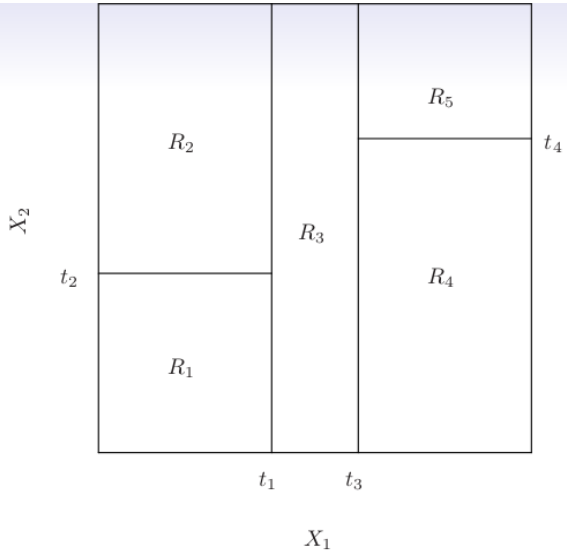
- We first select the predictor X_j and the cutpoint s such that splitting the predictor space into the regions $\{X|Xj < s\}$ and $\{X|Xj \geq s\}$ leads to the greatest possible reduction in RSS.
- Next, we repeat the process, looking for the best predictor and best cutpoint in order to split the data further so as to minimize the RSS within each of the resulting regions.
- However, this time, instead of splitting the entire predictor space, we split one of the two previously identified regions. We now have three regions.
- Again, we look to split one of these three regions further, so as to minimize the RSS. The process continues until a stopping criterion is reached; for instance, we may continue until no region contains more than five observations.

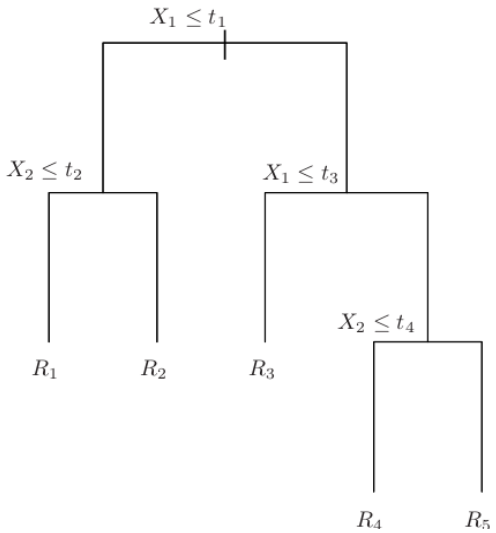
Predictions

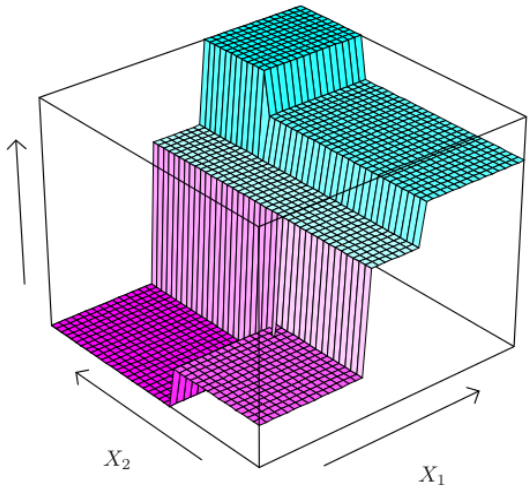
Predictions

- We predict the response for a given test observation using the mean of the training observations in the region to which that test observation belongs.
- A five-region example of this approach is shown in the next slide.









Pruning a tree

Pruning a tree

- The process described above may produce good predictions on the training set, but is likely to *overfit* the data, leading to poor test set performance.
- A smaller tree with fewer splits (that is, fewer regions $R_1, \ldots, R_J$) might lead to lower variance and better interpretation at the cost of a little bias.
- One possible alternative to the process described above is to grow the tree only so long as the decrease in the RSS due to each split exceeds some (high) threshold.
- This strategy will result in smaller trees, but is too *short-sighted*: a seemingly worthless split early on in the tree might be followed by a very good split – that is, a split that leads to a large reduction in RSS later on.

Pruning a tree – continued

- A better strategy is to grow a very large tree T_0, and then *prune* it back in order to obtain a subtree
- *Cost complexity pruning* – also known as *weakest link pruning* – is used to do this
- we consider a sequence of trees indexed by a nonnegative tuning parameter γ. For each value of γ there corresponds a subtree $T \subset T_0$ such that

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

is as small as possible. Here $|T|$ indicates the number of terminal nodes of the tree T, R_m is the rectangle (i.e. the subset of predictor space) corresponding to the m-th terminal node, and $\hat{y}_{R_m}$ is the mean of the training observations in R_m.

Choosing the best subtree

Choosing the best subtree

- The tuning parameter γ controls a trade-off between the subtree's complexity and its fit to the training data.
- We select an optimal value $\hat{\gamma}$ using cross-validation.
- We then return to the full data set and obtain the subtree corresponding to $\hat{\gamma}$.

Summary: tree algorithm

Summary: tree algorithm

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 γ.
3. Use K-fold cross-validation to choose γ. For each $k = 1, \ldots, K$:
   1. Repeat Steps 1 and 2 on the $\frac{K-1}{K}$th fraction of the training data, excluding the kth fold.
   2. Evaluate the mean squared prediction error on the data in the left-out kth fold, as a function of γ.

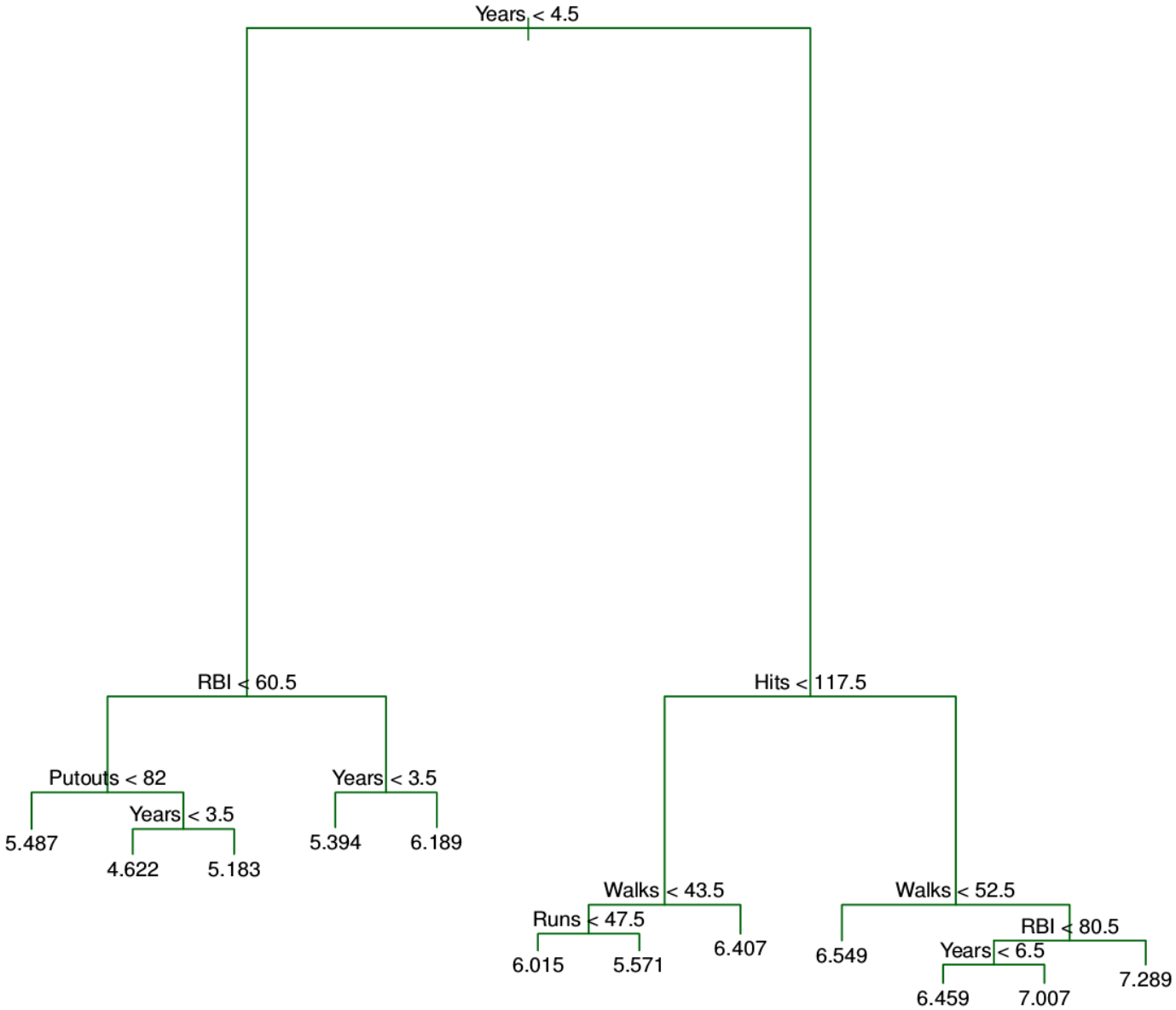
   Average the results, and pick γ to minimize the average error.
4. Return the subtree from Step 2 that corresponds to the chosen value of γ.

Baseball example continued

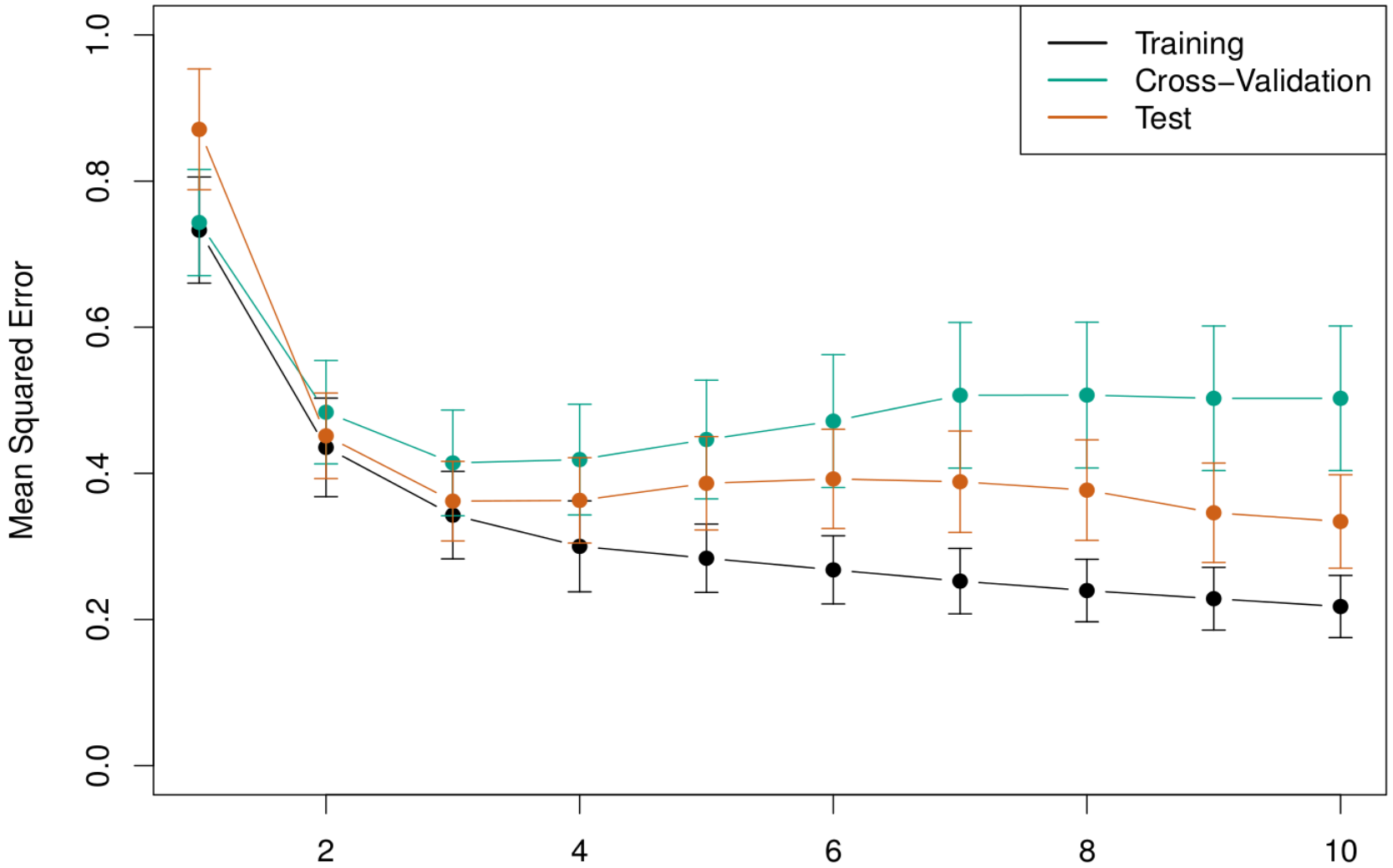
Baseball example continued

- First, we randomly divided the data set in half, yielding 132 observations in the training set and 131 observations in the test set.
- We then built a large regression tree on the training data and varied γ in in order to create subtrees with different numbers of terminal nodes.
- In addition we performed cross-validation in order to estimate the cross-validated MSE of the trees as a function of γ.

Baseball example continued



Baseball example continued



Classification Trees

Classification Trees

- Very similar to a regression tree, except that it is used to predict a qualitative response rather than a quantitative one.
- For a classification tree, we predict that each observation belongs to the *most commonly occurring class* of training observations in the region to which it belongs (majority vote).

Details of classification trees

- Just as in the regression setting, we use recursive binary splitting to grow a classification tree.
- In the classification setting, RSS cannot be used as a criterion for making the binary splits
- A natural alternative to RSS is the *classification error rate*. This is simply the fraction of the training observations in that region that do not belong to the most common class:

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

- Here $\hat{p}_{mk}$ represents the proportion of training observations in the mth region that are from the kth class.
- However classification error is not sufficiently sensitive for tree-growing, and in practice two other measures are preferable.

Gini index and Cross-Entropy

- The *Gini* index is defined by

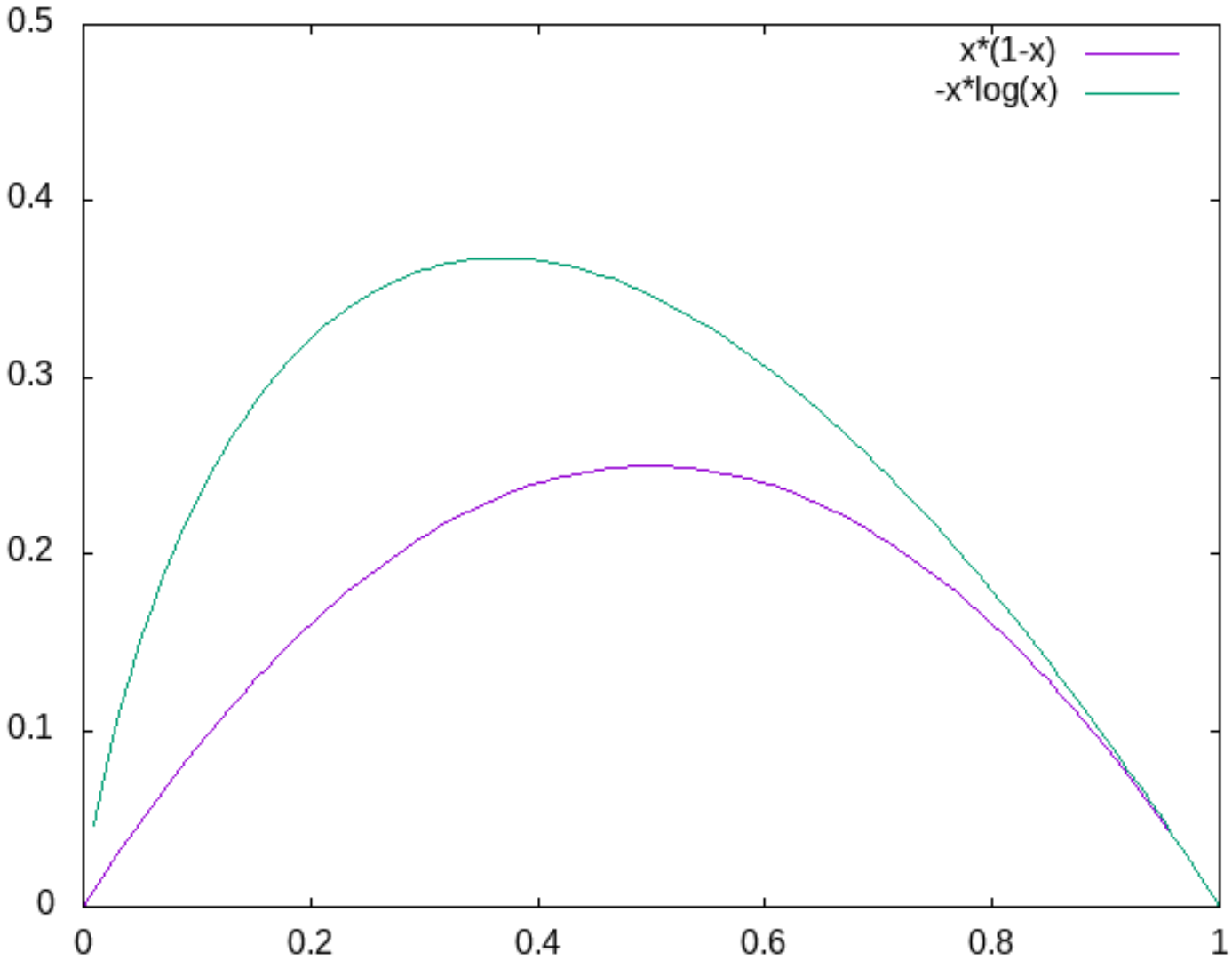
$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

  a measure of total variance across the K classes. The Gini index takes on a small value if all of the $\hat{p}_{mk}$'s are close to zero or one.
- For this reason the Gini index is referred to as a measure of node *purity* – a small value indicates that a node contains predominantly observations from a single class.
- An alternative to the Gini index is *cross-entropy*, given by

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

- It turns out that the Gini index and the cross-entropy are very similar numerically.

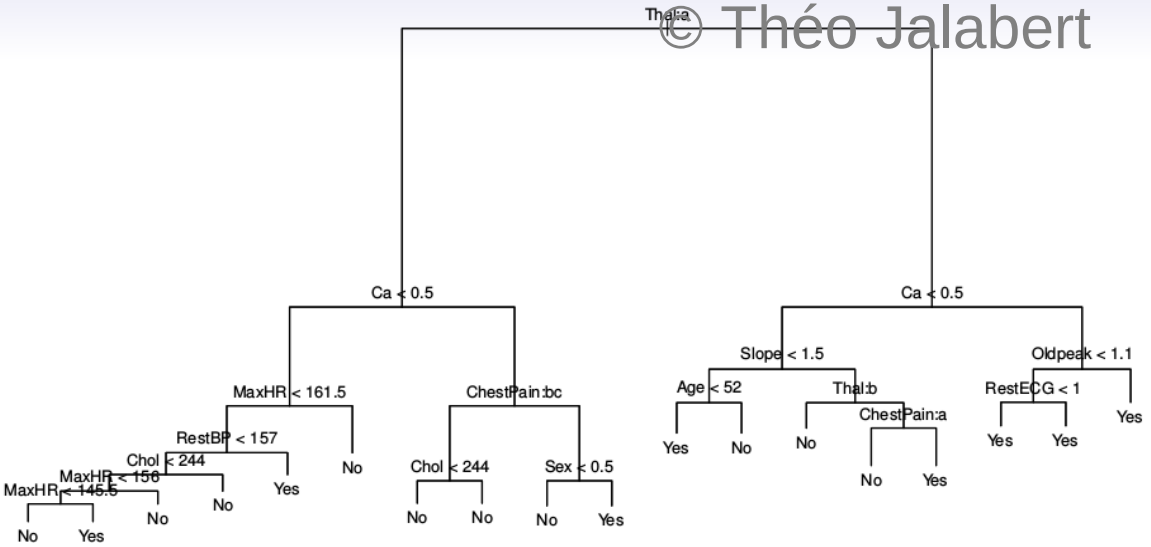
Gini index and Cross-Entropy

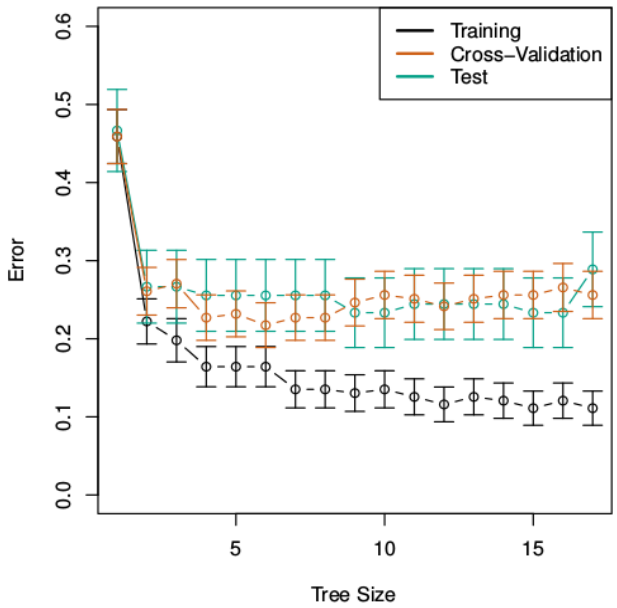


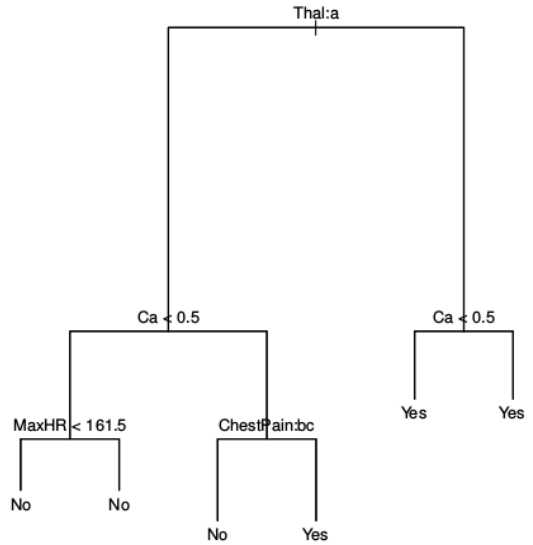
Example: heart data

Example: heart data

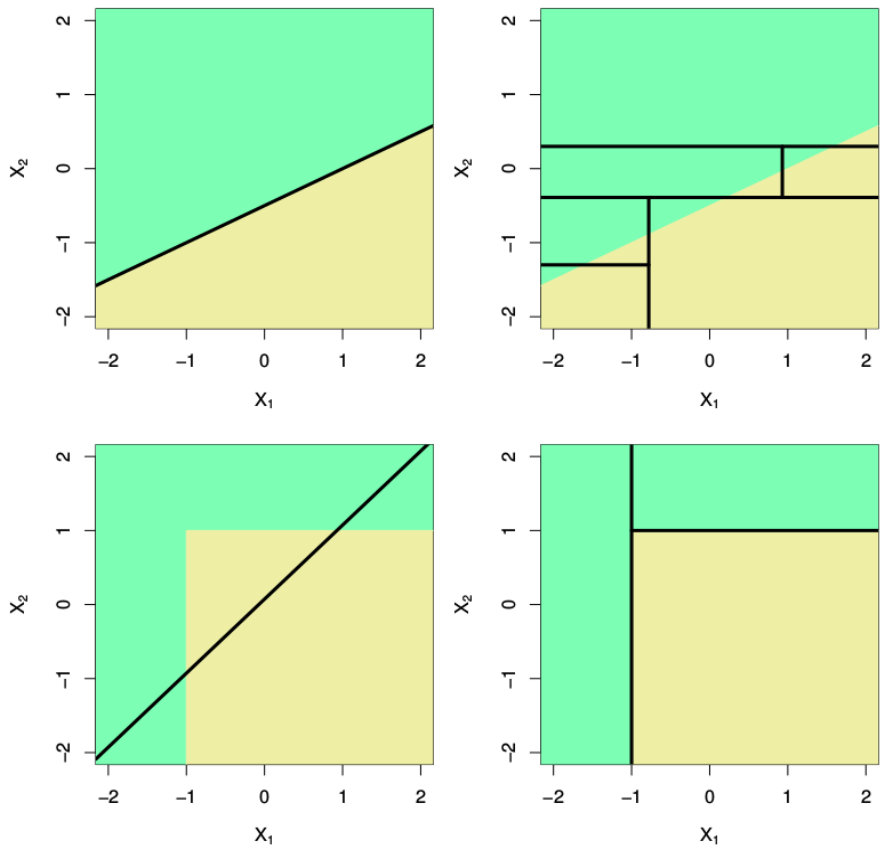
- These data contain a binary outcome `AHD` for 303 patients who presented with chest pain.
- An outcome value of `Yes` indicates the presence of heart disease based on an angiographic test, while `No` means no heart disease.
- There are 13 predictors including `Age`, `Sex`, `Chol` (a cholesterol measurement), and other heart and lung function measurements.
- Cross-validation yields a tree with six terminal nodes. See next figure.







Trees Versus Linear Models



Top Row: True linear boundary; Bottom row: true non-linear boundary. Left column: linear model; Right column: tree-based model

Advantages and Disadvantages of Trees

- pro Trees are very easy to explain to people. In fact, they are even easier to explain than linear regression!
- pro One might believe that decision trees more closely mirror human decision-making than do the regression and classification approaches seen in previous lecture.
- pro Trees can be displayed graphically, and are easily interpreted even by a non-expert (especially if they are small).
- pro Trees can easily handle qualitative predictors to some extend without the need to create dummy variables.
- con Unfortunately, trees generally do not have the same level of predictive accuracy as some of the other regression and classification approaches seen so far.

However, by aggregating many decision trees, the predictive performance of trees can be substantially improved.

Bagging

Bagging

- *Bootstrap aggregation*, or *bagging*, is a general-purpose procedure for reducing the variance of a statistical learning method; we introduce it here because it is particularly useful and frequently used in the context of decision trees.
- Recall that given a set of n independent observations $Z_1, \ldots, Z_n$, each with variance σ^2, the variance of the mean $\bar{Z}$ of the observations is given by σ^2/n.
- In other words, *averaging a set of observations reduces variance.* Of course, this is not practical because we generally do not have access to multiple training sets.

Bagging – continued

Bagging – continued

- Instead, we can bootstrap, by taking repeated samples from the (single) training data set.
- In this approach we generate B different bootstrapped training data sets. We then train our method on the b-th bootstrapped training set in order to get $\hat{f}^{*b}(x)$, the prediction at a point x. We then average all the predictions to obtain

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

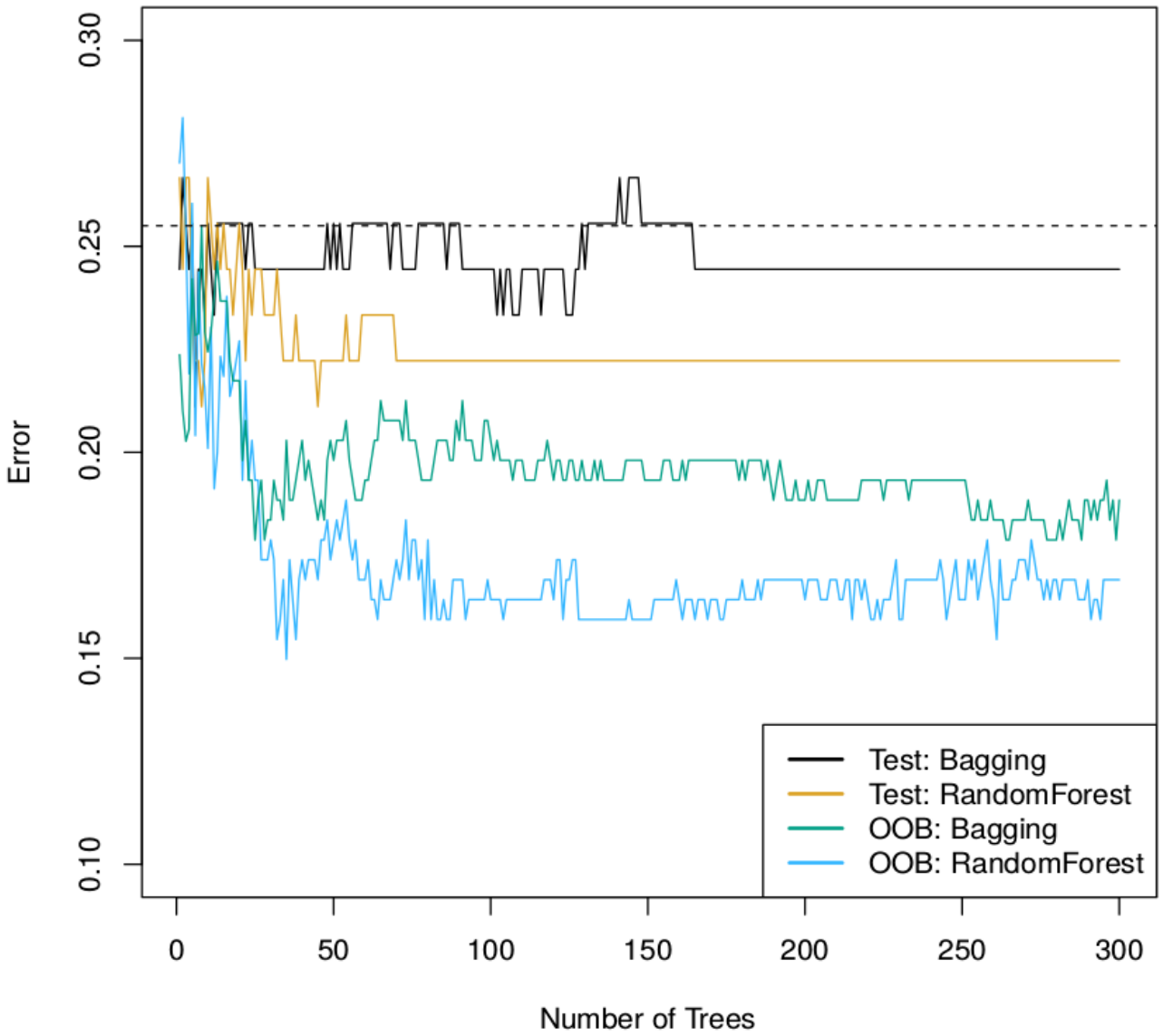
This is called bagging.

Bagging classification trees

Bagging classification trees

- The above prescription applies to regression trees
- For classification trees: for each test observation, we record the class predicted by each of the B trees, and take a *majority vote*: the overall prediction is the most commonly occurring class among the B predictions.

Bagging the heart data



Details of previous figure

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

Out-of-Bag Error Estimation

Out-of-Bag Error Estimation

- It turns out that there is a very straightforward way to estimate the test error of a bagged model.
- Recall that the key to bagging is that trees are repeatedly fit to bootstrapped subsets of the observations. One can show that on average, each bagged tree makes use of around two-thirds of the observations.
- The remaining one-third of the observations not used to fit a given bagged tree are referred to as the *out-of-bag* (OOB) observations.
- We can predict the response for the ith observation using each of the trees in which that observation was OOB. This will yield around $B/3$ predictions for the ith observation, which we average.
- This estimate is essentially the LOO cross-validation error for bagging, if B is large.

Random Forests

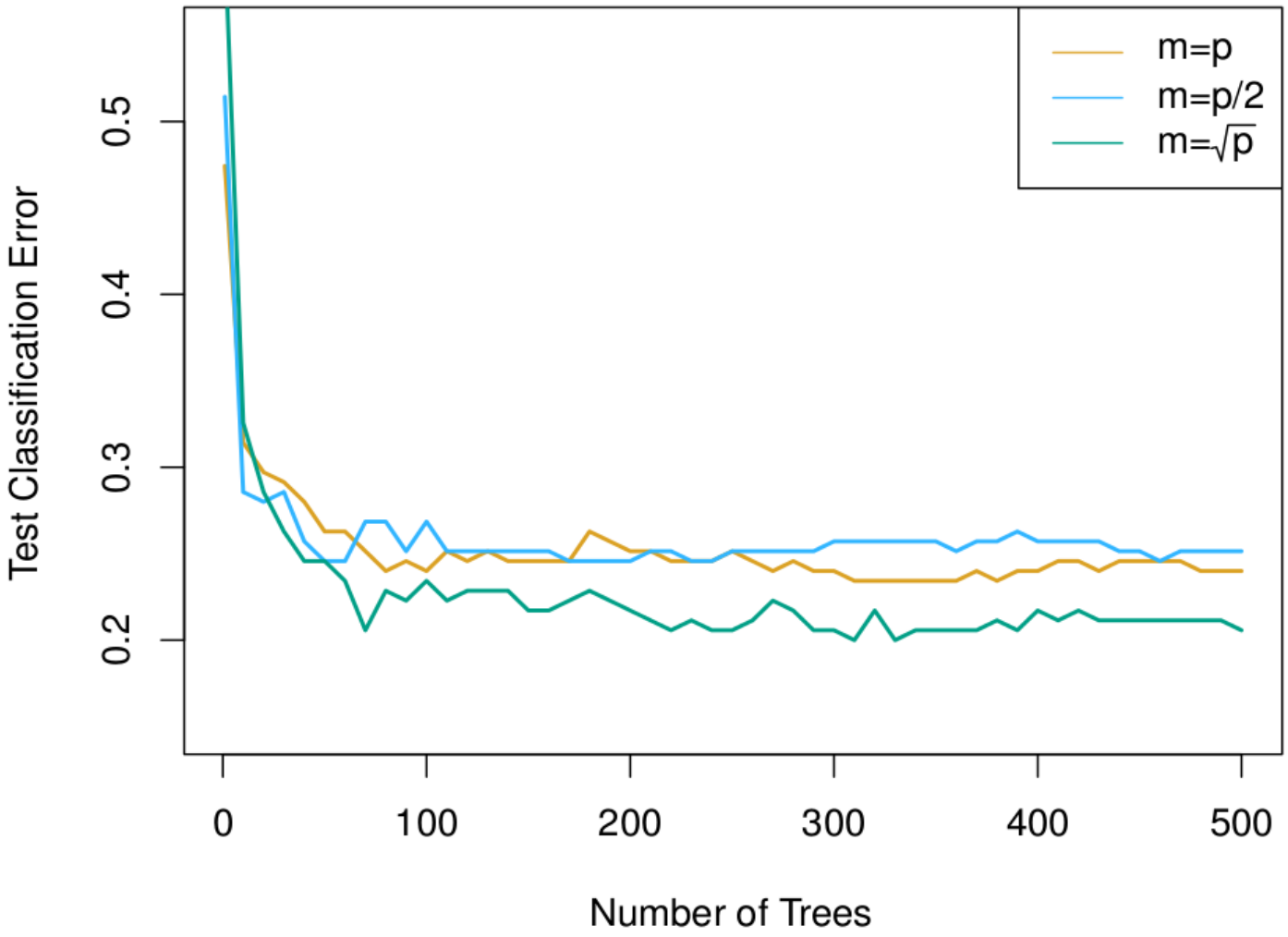
Random Forests

- Random forests provide an improvement over bagged trees by way of a small tweak that *decorrelates* the trees. This reduces the variance when we average the trees.
- As in bagging, we build a number of decision trees on bootstrapped training samples.
- But when building these decision trees, each time a split in a tree is considered, *a random selection of m predictors* is chosen as split candidates from the full set of p predictors. The split is allowed to use only one of those m predictors.
- A fresh selection of m predictors is taken at each split, and typically we choose $m \approx \sqrt{p}$ – that is, the number of predictors considered at each split is approximately equal to the square root of the total number of predictors (4 out of the 13 for the Heart data).

Example: gene expression data © Théo Jalabert

- Applying random forests to a high-dimensional biological data set consisting of expression measurements of 4,718 genes measured on tissue samples from 349 patients.
- There are around 20,000 genes in humans, and individual genes have different levels of activity, or expression, in particular cells, tissues, and biological conditions.
- Each of the patient samples has a qualitative label with 15 different levels: either normal or one of 14 different types of cancer.
- We use random forests to predict cancer type based on the 500 genes that have the largest variance in the training set.
- We randomly divided the observations into a training and a test set, and applied random forests to the training set for three different values of the number of splitting variables m.

Results: gene expression data



Details of previous figure

Details of previous figure

- Results from random forests for the fifteen-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

- Like bagging, boosting is a general approach that can be applied to many statistical learning methods for regression or classification. We only discuss boosting for decision trees.
- Recall that bagging involves creating multiple copies of the original training data set using the bootstrap, fitting a separate decision tree to each copy, and then combining all of the trees in order to create a single predictive model.
- Notably, each tree is built on a bootstrap data set, independent of the other trees.
- Boosting works in a similar way, except that the trees are grown *sequentially*: each tree is grown using information from previously grown trees.

Boosting algorithm 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:
   - 2.1 Fit a tree $\hat{f}^b$ with d splits ($d+1$ terminal nodes) to the training data (X, r).
   - 2.2 Update $\hat{f}$ by adding in a shrunken version of the new tree:

$$\hat{f}(x) \leftarrow \hat{f}(x) + \eta \hat{f}^b(x).$$

   - 2.3 Update the residuals

$$r_i \leftarrow r_i - \eta \hat{f}^b(x_i).$$

3. Output the boosted model,

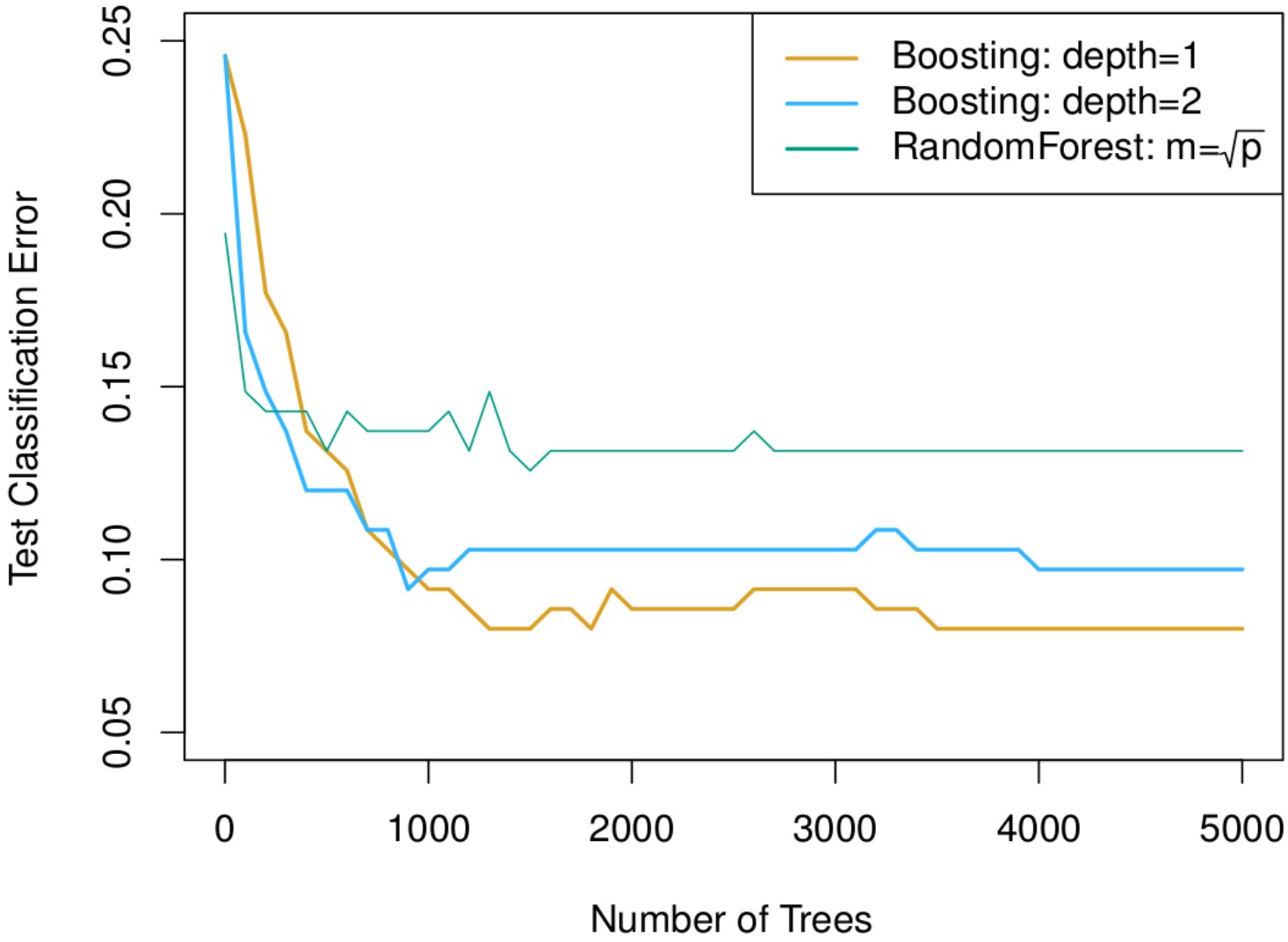
$$\hat{f}(x) = \sum_{b=1}^{B} \eta \hat{f}^b(x)$$

What is the idea behind this procedure?

What is the idea behind this procedure?

- Unlike fitting a single large decision tree to the data, which amounts to *fitting the data hard* and potentially overfitting, the boosting approach instead *learns slowly*.
- Given the current model, we fit a decision tree to the residuals from the model. We then add this new decision tree into the fitted function in order to update the residuals.
- Each of these trees can be rather small, with just a few terminal nodes, determined by the parameter d in the algorithm.
- By fitting small trees to the residuals, we slowly improve $\hat{f}$ in areas where it does not perform well. The shrinkage parameter η slows the process down even further, allowing more and different shaped trees to attack the residuals.

Gene expression data continued



Details of previous figure

Details of previous figure

- Results from performing boosting and random forests on the fifteen-class gene expression data set in order to predict cancer versus normal.
- The test error is displayed as a function of the number of trees. For the two boosted models, $\eta = 0.01$. Depth-1 trees slightly outperform depth-2 trees, and both outperform the random forest, although the standard errors are around 0.02, making none of these differences significant.
- The test error rate for a single tree is 24%.

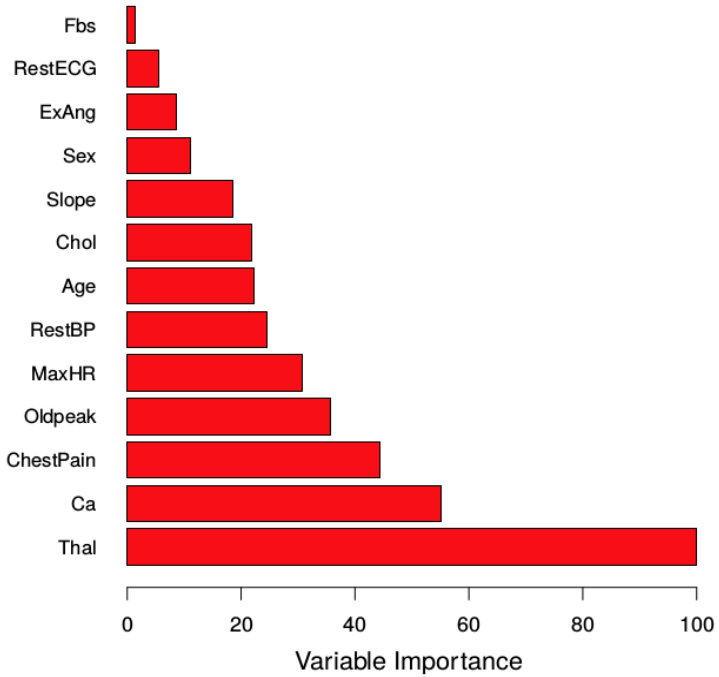
Tuning parameters for boosting

Tuning parameters for boosting

1. The *number of trees* B. Unlike bagging and random forests, boosting can overfit if B is too large, although this overfitting tends to occur slowly if at all. We use cross-validation to select B.
2. The *shrinkage parameter* η, a small positive number. This controls the rate at which boosting learns. Typical values are $0.3, 0.01$ or 0.001, and the right choice can depend on the problem. Very small η can require using a very large value of B in order to achieve good performance.
3. The *number of splits* d in each tree, which controls the complexity of the boosted ensemble. Often $d = 1$ works well, in which case each tree is a *stump*, consisting of a single split and resulting in an additive model. More generally d is the *interaction depth*, and controls the interaction order of the boosted model, since d splits can involve at most d variables.

Variable importance measure

- For bagged/RF regression trees, we record the total amount that the RSS is decreased due to splits over a given predictor, averaged over all B trees. A large value indicates an important predictor.
- Similarly, for bagged/RF classification trees, we add up the total amount that the Gini index is decreased by splits over a given predictor, averaged over all B trees.



Summary

Summary

- Decision trees are simple and interpretable models for regression and classification
- However they are often not competitive with other methods in terms of prediction accuracy
- Bagging, random forests and boosting are good methods for improving the prediction accuracy of trees. They work by growing many trees on the training data and then combining the predictions of the resulting ensemble of trees.
- The latter two methods – random forests and boosting – are among the state-of-the-art methods for supervised learning. However their results can be difficult to interpret.

Software packages

XGBoost

`https://github.com/dmlc/xgboost`

dmlc

XGBoost eXtreme Gradient Boosting

build passing | docs latest | license Apache 2.0 | CRAN 0.4-3 | pypi package 0.4a30 | gitter join chat

Documentation | Resources | Installation | Release Notes | RoadMap

XGBoost is an optimized distributed gradient boosting library designed to be highly ***efficient***, ***flexible*** and ***portable***. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting(also known as GBDT, GBM) that solve many data science problems in a fast and accurate way. The same code runs on major distributed environment(Hadoop, SGE, MPI) and can solve problems beyond billions of examples.

XGBoost Implementation

Optimizing the tree structure

For $\hat{y} = \sum_{b=1}^{B} f_b$ and a general loss function l, minimize the objective

$$\min \leftarrow \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{b=1}^{B} \Omega(f_b)$$

with regularization $\Omega(f_b) = \gamma|T| + \frac{1}{2}\lambda \sum_{j=1}^{|T|} w_j^2$, where $|T| = d + 1$ is the number of leafs and w_j is the regression weight of leaf (region) R_j

XGBoost Implementation

Additive Training (Boosting)

Start from constant prediction, add a new function each time (for simplicity we set $\eta = 1$ for now)

$$
\begin{aligned}
\hat{y}_i^{(0)} &= 0 \\
\hat{y}_i^{(1)} &= f_1(x_i) = \hat{y}^{(0)} + f_1(x_i) \\
\hat{y}_i^{(2)} &= f_1(x_i) + f_2(x_i) = \hat{y}^{(1)} + f_2(x_i) \\
&\dots \\
\hat{y}_i^{(t)} &= \sum_{b=1}^{t} f_b(x_i) = \hat{y}^{(t-1)} + f_t(x_i)
\end{aligned}
$$

XGBoost Implementation

- How do we decide which f_t (splits) to add? Optimize the objective!
- The prediction at round t is $\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i)$, so that

$$\begin{aligned} \text{Obj}^{(t)} &= \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t)}) + \sum_{b=1}^{t} \Omega(f_b) \\ &= \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) + C \end{aligned}$$

- Consider l_2-loss:

$$\begin{aligned} \text{Obj}^{(t)} &= \sum_{i=1}^{n} \left(y_i - (\hat{y}_i^{(t-1)} + f_t(x_i))\right)^2 + \Omega(f_t) + C_1 \\ &= \sum_{i=1}^{n} \left[2(\hat{y}_i^{(t-1)} - y_i) f_t(x_i) + f_t(x_i)^2\right] + \Omega(f_t) + C_2 \end{aligned}$$

XGBoost Implementation

Taylor Expansion for general loss

- Take Taylor expansion of the objective
  - Recall $f(x+\Delta x) \approx f(x) + f'(x)\Delta x + \frac{1}{2}f''(x)\Delta x^2$
  - Define $g_i = \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)}), \quad h_i = \partial^2_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$
- Then

$$\text{Obj}^{(t)} \approx \sum_{i=1}^{n} \left[l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2 \right] + \Omega(f_t) + C$$

- In case of l_2-loss:

$$\begin{aligned} g_i &= \partial_{\hat{y}_i^{(t-1)}} (\hat{y}_i^{(t-1)} - y_i)^2 = 2(\hat{y}_i^{(t-1)} - y_i) \\ h_i &= \partial^2_{\hat{y}_i^{(t-1)}} (\hat{y}_i^{(t-1)} - y_i)^2 = 2 \end{aligned}$$

XGBoost Implementation

Revisit the objectives

- Define $J : \mathbb{R}^p \to T, x \mapsto R_j$ for $j : x \in R_j$
- Remove the constants and regroup by leafs

$$\begin{aligned}
\text{Obj}^{(t)} &\approx \sum_{i=1}^{n} \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t) \\
&= \sum_{i=1}^{n} \left[g_i w_{J(x_i)} + \frac{1}{2} h_i w_{J(x_i)}^2 \right] + \gamma |T| + \frac{1}{2} \lambda \sum_{j=1}^{|T|} w_j^2 \\
&= \sum_{j=1}^{|T|} \left[\Big(\sum_{i:x_i \in R_j} g_i \Big) w_j + \frac{1}{2} \Big(\sum_{i:x_i \in R_j} h_i + \lambda \Big) w_j^2 \right] + \gamma |T|
\end{aligned}$$

- This is sum of $|T|$ independent quadratic functions

XGBoost Implementation

- Two facts about single variable quadratic function

$$\arg\min_x Gx + \frac{1}{2}Hx^2 = -\frac{G}{H}, H > 0, \quad \min_x Gx + \frac{1}{2}Hx^2 = -\frac{1}{2}\frac{G^2}{H}$$

- Let us define $G_j = \sum_{i:x_i \in R_j} g_i$ and $H_j = \sum_{i:x_i \in R_j} h_i$ then

$$\text{Obj}^{(t)} \approx \sum_{j=1}^{|T|} \left[G_j w_j + \frac{1}{2}(H_j + \lambda) w_j^2 \right] + \gamma |T|$$

- Assume the structure of tree $(R_j)_{j=1}^{|T|}$ is fixed, the optimal weight in each leaf and the resulting objective value are

$$w_j^* = -\frac{G_j}{H_j + \lambda}, \qquad \text{Obj}^{(t)} \approx -\frac{1}{2} \sum_{j=1}^{|T|} \frac{G_j^2}{H_j + \lambda} + \gamma |T|$$

XGBoost Implementation

Greedy Learning of the Tree

In practice, we grow the tree greedily

- Start from tree with depth 0
- For each leaf node of the tree, try to add a split. The change of objective after adding the split is

$$\text{Gain} = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right] - \gamma$$

- Left to right linear scan over sorted instance is enough to decide the best split along the feature
- Stop splitting if the best split has negative gain

XGBoost Parameters

- `eta` [default=0.3, alias: `learning_rate`]
  Step size shrinkage used in update to prevents overfitting. After each boosting step, we can directly get the weights of new features, and `eta` shrinks the feature weights to make the boosting process more conservative.
- `gamma` [default=0, alias: `min_split_loss`]
  Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger `gamma` is, the more conservative the algorithm will be.
- `max_depth` [default=6]
  Maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit. 0 indicates no limit.
- `colsample_bytree` [default=1]
  Subsample ratio of columns when constructing each tree. Subsampling will occur once in every boosting iteration.

XGBoost Parameters – cont'd

- `lambda` [default=1, alias: `reg_lambda`]
  l_2 regularization term on weights. Increasing this value will make model more conservative.
- `num_round`
  The number of rounds for boosting (B).
- `objective` [default=`reg:squarederror`] (selection)
  - `reg:squarederror`: regression using l_2 loss
  - `reg:logistic`: logistic regression
  - `binary:logistic`: logistic regression for binary classification, output probability
  - `multi:softmax`: set XGBoost to do multiclass classification using the softmax objective
  - `rank:pairwise`: Use LambdaMART to perform pairwise ranking where the pairwise loss is minimized
  - `rank:ndcg`: Use LambdaMART to perform list-wise ranking where Normalized Discounted Cumulative Gain (NDCG) is maximized

Further gradient boosting libraries

Further gradient boosting libraries

LightGBM by Microsoft, `https://github.com/Microsoft/LightGBM`

- Introduced a faster histogram based split search procedure
- provided GPU implementation

Both features are now also available with XGBoost (`tree_method = hist`)

CatBoost by Yandex, `https://github.com/catboost/catboost`

- Improved handling of categorial features