Statistical Methods for Data Science

Semester 1 2021

DATA7202

Tree Models

Slava Vaisman

The University of Queensland

r.vaisman@uq.edu.au

1. Decision trees
   - Regional Prediction Functions
   - Splitting Rules
   - Termination Criterion
   - Basic implementation of a decision tree
   - Additional Considerations
   - Controlling the Tree Shape
   - Advantages Decision Trees

2. Bootstrap Aggregation and Random Forests

3. Boosting

Introduction

- Statistical learning methods based on decision trees have gained tremendous popularity due to their simplicity, intuitive representation, and predictive accuracy.
- In this unit, we give an introduction to the construction and use of such trees.
- We also discuss two key ensemble methods, namely bootstrap aggregation and boosting, which can further improve the efficiency of decision trees and other learning methods.

Decision trees

- Tree-based methods provide a simple, intuitive, and powerful mechanism for both regression and classification.
- The main idea is to divide a (potentially complicated) feature space $\mathcal{X}$ into smaller regions and fit a simple prediction function to each region.
- For example, in a regression setting, one could take the mean of the training responses associated with the training features that fall in that specific region.
- In the classification setting, a commonly used prediction function takes the majority vote among the corresponding response variables.

We start with a simple classification example.

Decision trees example (1)

The left panel of Figure 1 shows a training set of 15 two-dimensional points (features) falling into two classes (red and blue). How should the new feature vector (black point) be classified?

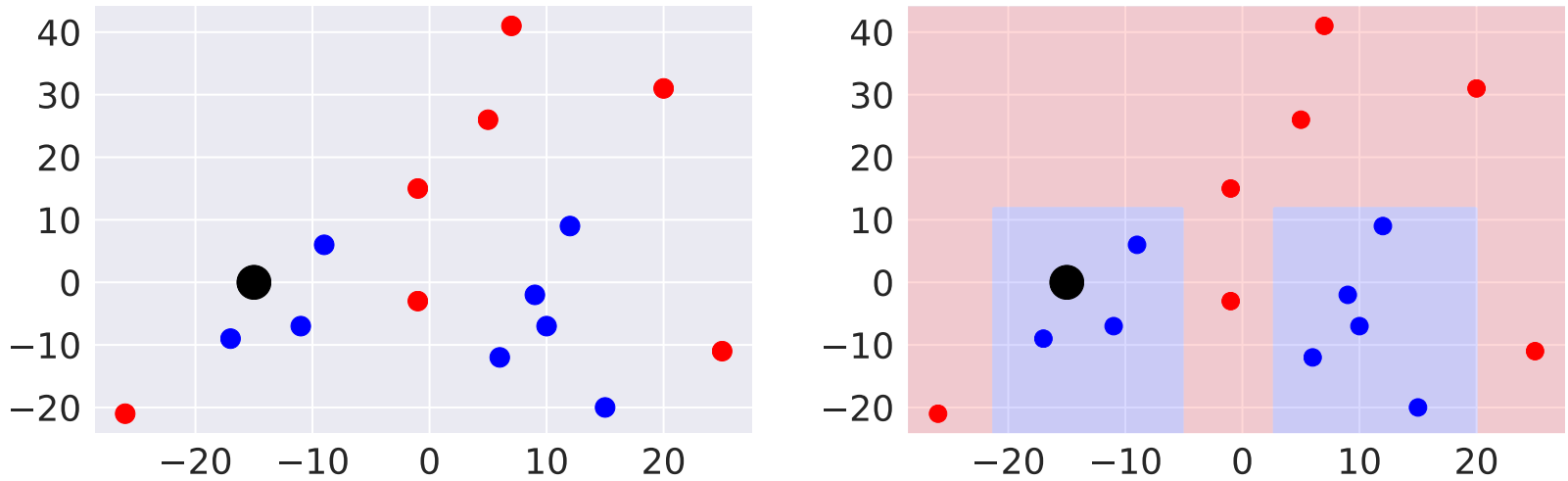
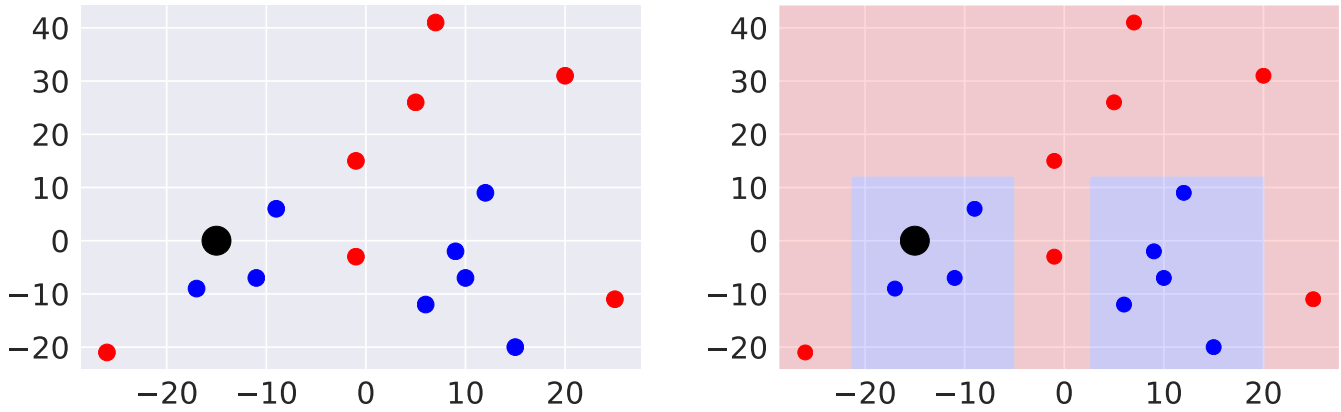


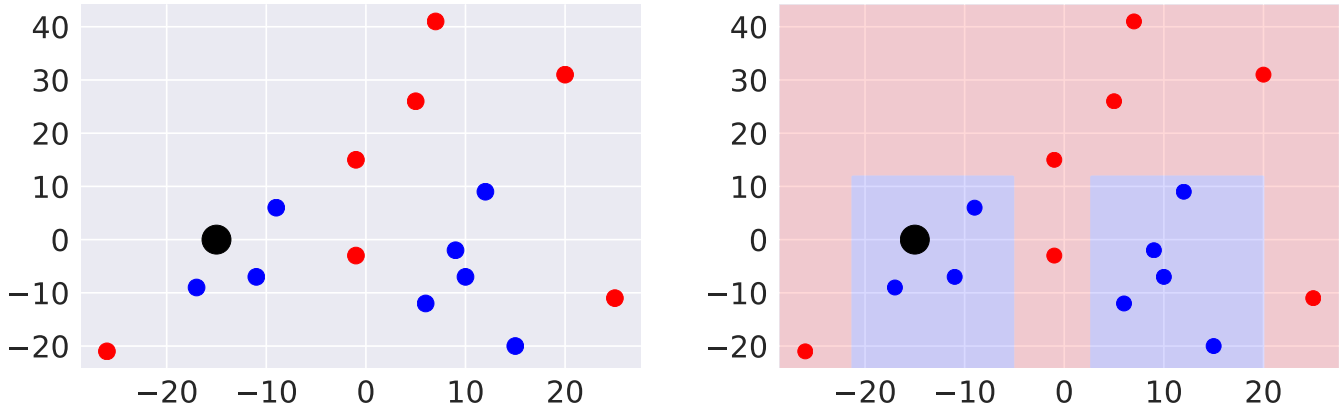
Figure 1: Left: training data and a new feature. Right: a partition of the feature space.

Decision trees example (2)

- It is not possible to linearly separate the training set, but we can partition the feature space $\mathcal{X} = \mathbb{R}^2$ into rectangular regions and assign a class (color) to each region, as shown in the right panel of Figure 1.
- Points in these regions are classified accordingly as blue or red.
- The partition thus defines a classifier (prediction function) g (in this unit we will use g instead of h), that assigns to each feature vector x a class "red" or "blue".



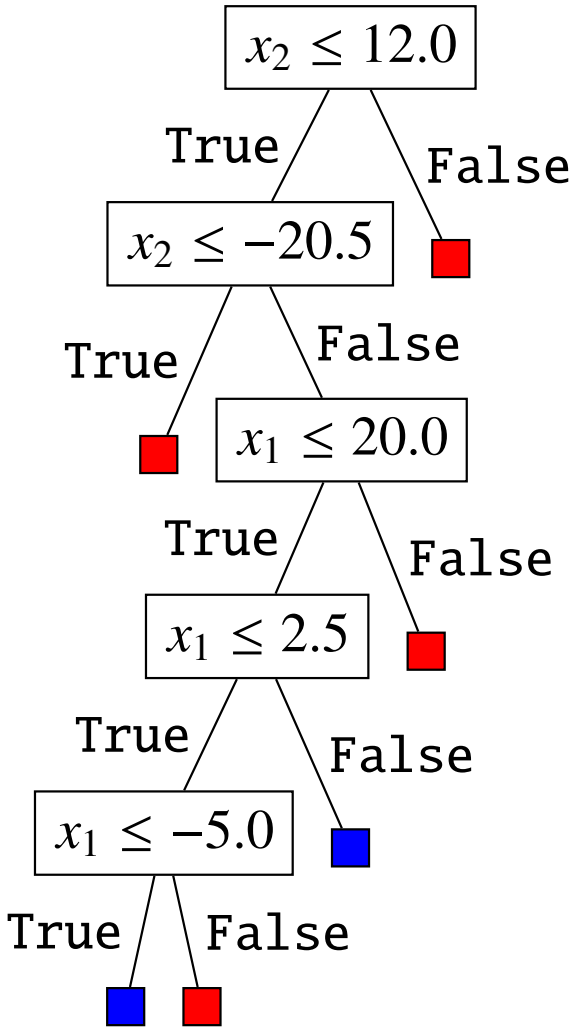
Decision trees example (3)



- For example, for $\mathrm{x} = [-15, 0]^\top$ (solid black point), $g(\mathrm{x}) =$ "blue", since it belongs to a blue region of the feature space.
- Both the classification procedure and the partitioning of the feature space can be conveniently represented by a binary decision tree. This is a tree where each node v corresponds to a region (subset) R_v of the feature space X — the root node corresponding to the feature space itself.

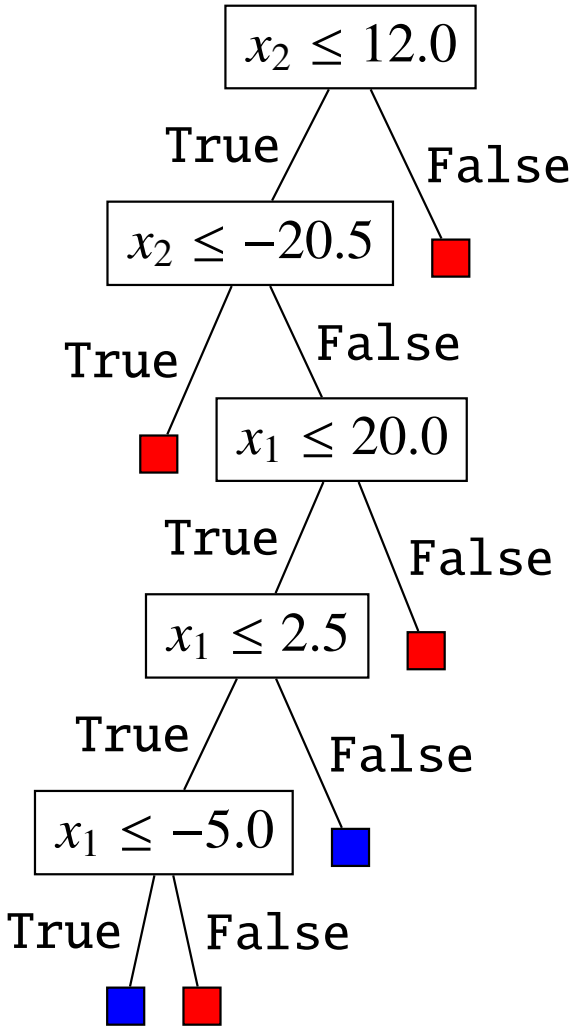
Decision trees example (4)

- Each internal node v contains a logical condition that divides R_v into two disjoint subregions.
- The leaf nodes (the terminal nodes of the tree) are not subdivided, and their corresponding regions form a partition of X, as they are disjoint and their union is X.
- Associated with each leaf node w is also a regional prediction function g^w on R_w.
- The partitioning of the 2D space, was obtained from the decision tree shown on the right.



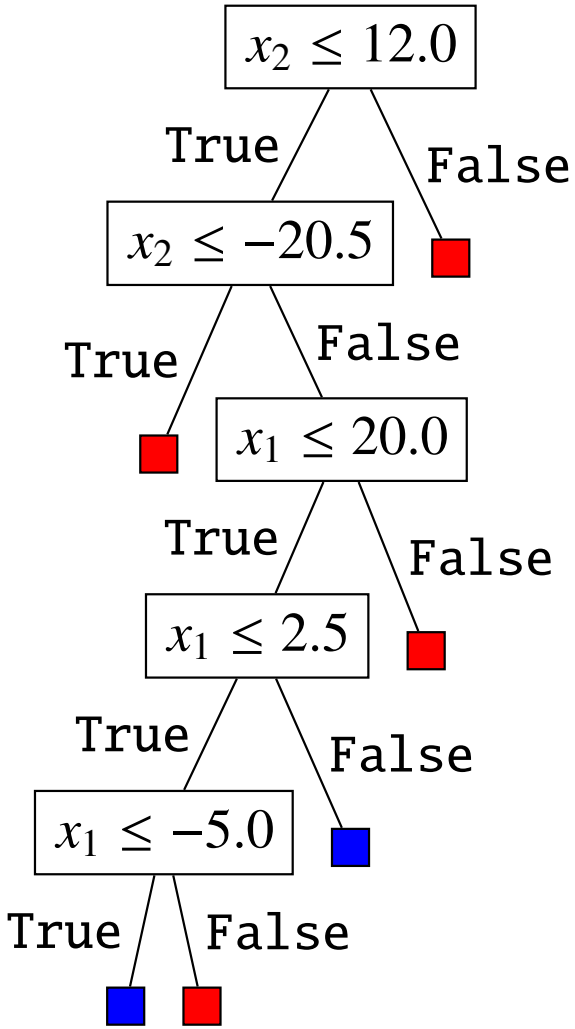
Decision trees example (5)

- As an illustration of the decision procedure, consider again the input $\mathrm{x} = [x_1, x_2]^\top = [-15, 0]^\top$.
- The classification process starts from the tree root, which contains the condition $x_2 \leq 12.0$.
- As the second component of x is 0, the root condition is satisfied and we proceed to the left child, which contains the condition $x_2 \leq -20.5$.



Decision trees example (6)

- The next step is similar. As $0 > -20.5$, the condition is not satisfied and we proceed to the right child.
- Such an evaluation of logical conditions along the tree path will eventually bring us to a leaf node and its associated region.
- In this case the process terminates in a leaf that corresponds to the left blue region in the right-hand panel of the Figure.



The general setting (1)

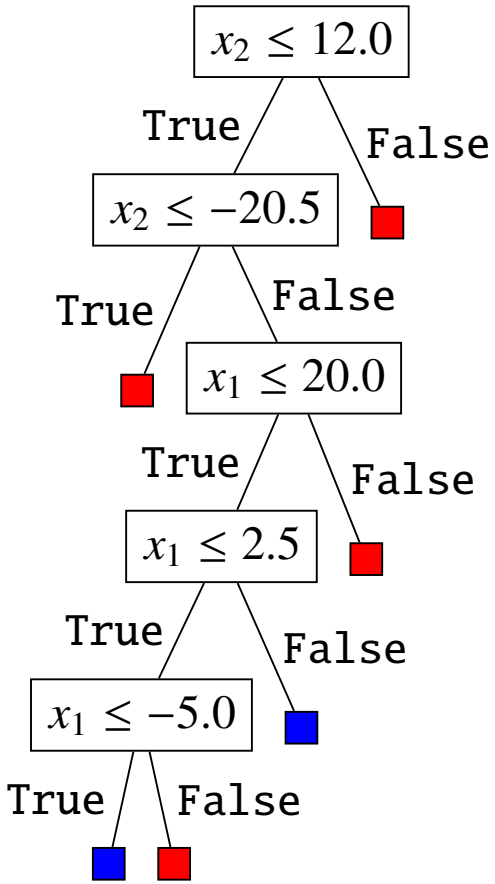
- More generally, a binary tree T will partition the feature space $\mathcal{X}$ into as many regions as there are leaf nodes.
- Denote the set of leaf nodes by $\mathcal{W}$. The overall prediction function g that corresponds to the tree can then be written as

$$g(x) = \sum_{w \in \mathcal{W}} g^w(x)\, \mathbb{1}\{x \in \mathcal{R}_w\}, \tag{1}$$

  where $\mathbb{1}$ denotes the indicator function.
- The representation (1) is very general and depends on
  1. how the regions $\{\mathcal{R}_w\}$ are constructed via the logical conditions in the decision tree, and
  2. how the regional prediction functions of the leaf nodes are defined.
- Simple logical conditions of the form $x_j \leq \xi$ split a Euclidean feature space into rectangles aligned with the axes.

The general setting (2)

For example, the below Figure partitions the feature space into six rectangles: two blue and four red rectangles.



Decision trees for classification and regression

In a classification setting:

- the regional prediction function g^w corresponding to a leaf node w takes values in the set of possible class labels.
- In most cases, as in our tree example, it is taken to be constant on the corresponding region R_w.

In a regression setting:

- g^w is real-valued and also usually takes only one value.
- That is, every feature vector in R_w leads to the same predicted value. Of course, different regions will usually have different predicted values.

Constructing a tree (1)

- Constructing a tree with a training set $\tau = \{(x_i, y_i)\}_{i=1}^n$ amounts to minimizing the training loss

$$\ell_\tau(g) = \frac{1}{n}\sum_{i=1}^{n} \text{Loss}(y_i, g(x_i)) \tag{2}$$

for some loss function.

- With g of the form $g(x) = \sum_{w \in W} g^w(x)\, \mathbb{1}\{x \in R_w\}$, we can write

$$\ell_\tau(g) = \frac{1}{n}\sum_{i=1}^{n} \text{Loss}(y_i, g(x_i)) = \frac{1}{n}\sum_{i=1}^{n}\sum_{w \in W} \mathbb{1}\{x_i \in R_w\}\, \text{Loss}(y_i, g(x_i))$$

$$= \sum_{w \in W} \underbrace{\frac{1}{n}\sum_{i=1}^{n} \mathbb{1}\{x_i \in R_w\}\, \text{Loss}(y_i, g^w(x_i))}_{(*)},$$

where $(*)$ is the contribution by the regional prediction function g^w to the overall training loss.

Constructing a tree (2)

- In the case where all $\{x_i\}$ are different, finding a decision tree T that gives a zero squared-error or zero–one training loss is easy (assignment?).
- but such an "overfitted" tree will have poor predictive behavior, expressed in terms of the generalization risk.
- Instead we consider a restricted class of decision trees and aim to minimize the training loss within that class.
- It is common to use a top-down greedy approach, which can only achieve an approximate minimization of the training loss.

Top-Down Construction of Decision Trees (1)

- Let $\tau = \{(x_i, y_i)\}_{i=1}^{n}$ be the training set.
- The key to constructing a binary decision tree T is to specify a *splitting rule* for each node v, which can be defined as a logical function $s : \mathcal{X} \to \{\texttt{False}, \texttt{True}\}$ or, equivalently, a binary function $s : \mathcal{X} \to \{0, 1\}$.
- For example, in the decision tree that we discussed earlier, the root node has splitting rule $x \mapsto \mathbb{1}\{x_1 \leq 12.0\}$, in correspondence with the logical condition $\{x_1 \leq 12.0\}$.
- During the construction of the tree, each node v is associated with a specific region $R_v \subseteq \mathcal{X}$ and therefore also the training subset $\{(x, y) \in \tau : x \in R_v\} \subseteq \tau$.

Top-Down Construction of Decision Trees (2)

Using a splitting rule s, we can divide any subset σ of the training set τ into two sets:

$$\sigma_{\mathrm{T}} := \{(\mathsf{x}, y) \in \sigma \ : \ s(\mathsf{x}) = \texttt{True}\}$$

$$\sigma_{\mathrm{F}} := \{(\mathsf{x}, y) \in \sigma \ : \ s(\mathsf{x}) = \texttt{False}\}.$$

- Starting from an empty tree and the initial data set τ, a generic decision tree construction takes the form of the recursive Algorithm in the next slide.
- Here we use the notation T_v for a subtree of T starting from node v.
- The final tree T is thus obtained via

$$T = \text{Construct\_Subtree}(v_0, \tau),$$

where v_0 is the root of the tree.

Top-Down Construction of Decision Trees (3)

Algorithm 1: `Construct_Subtree`

Input: A node v and a subset of the training data: $\sigma \subseteq \tau$.

Output: A (sub) decision tree T_v.

if termination criterion is met **then** `// v is a leaf node`

  Train a regional prediction function g^v using the training data σ.

else `// split the node`

  Find the best splitting rule s_v for node v.

  Create successors v_{T} and v_{F} of v.

  $\sigma_{\mathrm{T}} \leftarrow \{(\mathrm{x}, y) \in \sigma \ : \ s_v(\mathrm{x}) = \mathtt{True}\}$

  $\sigma_{\mathrm{F}} \leftarrow \{(\mathrm{x}, y) \in \sigma \ : \ s_v(\mathrm{x}) = \mathtt{False}\}$

  $T_{v_{\mathrm{T}}} \leftarrow \mathtt{Construct\_Subtree}\,(v_{\mathrm{T}}, \sigma_{\mathrm{T}})$ `// left branch`

  $T_{v_{\mathrm{F}}} \leftarrow \mathtt{Construct\_Subtree}\,(v_{\mathrm{F}}, \sigma_{\mathrm{F}})$ `// right branch`

end

return T_v

Top-Down Construction of Decision Trees (4)

- The splitting rule s_v divides the region R_v into two disjoint parts, say R_{v_T} and R_{v_F}.
- The corresponding prediction functions, g^{v_T} and g^{v_F}, satisfy

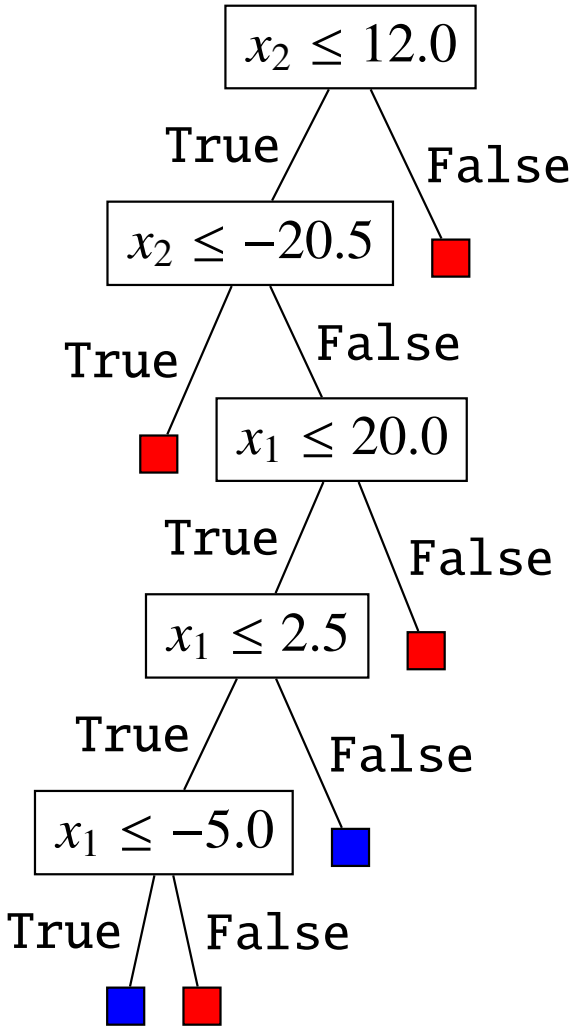
$$g^v(x) = g^T(x)\,\mathbb{1}\{x \in R_{v_T}\} + g^F(x)\,\mathbb{1}\{x \in R_{v_F}\}, \quad x \in \mathbb{R}_v.$$

In order to implement the procedure described in the tree construction algorithm, we need to address the following issues.

- The construction of the regional prediction functions g^v at the leaves (Line 2).
- The specification of the splitting rule (Line 4).
- The specification of the termination criterion (Line 1).

Regional Prediction Functions

- In general, there is no restriction on how to choose the prediction function g^w for a leaf node $v = w$ in Line 2 of the tree construction Algorithm.
- In principle we can train any model from the data; e.g., via linear regression. However, in practice very simple prediction functions are used.



Regional Prediction Functions — classification problems

- In the *classification* setting with class labels $0, \ldots, c-1$, the regional prediction function g^w for leaf node w is usually chosen to be *constant* and equal to the most common class label of the training data in the associated region R_w (ties can be broken randomly).
- More precisely, let n_w be the number of feature vectors in region R_w and let

$$p_z^w = \frac{1}{n_w} \sum_{\{(x,y)\in\tau \,:\, x\in R_w\}} \mathbb{1}_{\{y=z\}},$$

be the proportion of feature vectors in R_w that have class label $z = 0, \ldots, c-1$. The regional prediction function for node w is chosen to be the constant

$$g^w(x) = \operatorname*{argmax}_{z\in\{0,\ldots,c-1\}} p_z^w. \tag{3}$$

Regional Prediction Functions — regression problems

- In the *regression* setting, g^w is usually chosen as the mean response in the region; that is,

$$g^w(x) = \overline{y}_{R_w} := \frac{1}{n_w} \sum_{\{(x,y)\in\tau \,:\, x\in R_w\}} y, \tag{4}$$

  where n_w is again the number of feature vectors in R_w.
- It is not difficult to show that $g^v(x) = \overline{y}_{R_w}$ minimizes the squared-error loss with respect to all constant functions, in the region R_w (assignment?).

Splitting Rules (1)

- In Line 4 of the construction Algorithm, we divide region R_v into two sets, using a splitting rule (function) s_v.
- Consequently, the data set σ associated with node v (that is, the subset of the original data set τ whose feature vectors lie in R_v), is also split — into $\sigma_{\mathtt{T}}$ and $\sigma_{\mathtt{F}}$.
- What is the benefit of such a split in terms of a reduction in the training loss? If v were set to a leaf node, its contribution to the training loss would be:

$$\frac{1}{n}\sum_{i=1}^{n}\mathbb{1}_{\{(\mathrm{x},y)\in\sigma\}}\,\mathrm{Loss}(y_i, g^{v}(\mathrm{x}_i)).$$

If v were to be split instead, its contribution to the overall training loss would be:

$$\frac{1}{n}\sum_{i=1}^{n}\mathbb{1}_{\{(\mathrm{x},y)\in\sigma_{\mathtt{T}}\}}\,\mathrm{Loss}(y_i, g^{\mathtt{T}}(\mathrm{x}_i)) + \frac{1}{n}\sum_{i=1}^{n}\mathbb{1}_{\{(\mathrm{x},y)\in\sigma_{\mathtt{F}}\}}\,\mathrm{Loss}(y_i, g^{\mathtt{F}}(\mathrm{x}_i)).$$

Splitting Rules (2)

- Here, g^{T} and g^{F} are the prediction functions belonging to the child nodes v_{T} and v_{F}.
- A *greedy* heuristic is to pretend that the tree construction algorithm immediately terminates after the split, in which case v_{T} and v_{F} are leaf nodes, and g^{T} and g^{F} are readily evaluated.
- As an example, suppose the feature space is $\mathsf{X} = \mathbb{R}^p$ and we consider splitting rules of the form

$$s(\mathsf{x}) = \mathbb{1}\{x_j \leq \xi\}, \tag{5}$$

  for some $1 \leq j \leq p$ and $\xi \in \mathbb{R}$, where we identify 0 with `False` and 1 with `True`.
- Due to the computational and interpretative simplicity, such binary splitting rules are implemented in many software packages and are considered to be the *de facto* standard.

Splitting Rules — the regression setting

- For a regression problem, using a squared-error loss and a constant regional prediction function, we have

$$\frac{1}{n} \sum_{(\mathrm{x},y)\in\tau:x_j\leq\xi} (y-\overline{y}_{\mathtt{T}})^2 + \frac{1}{n} \sum_{(\mathrm{x},y)\in\tau:x_j>\xi} (y-\overline{y}_{\mathtt{F}})^2 , \tag{6}$$

  where $\overline{y}_{\mathtt{T}}$ and $\overline{y}_{\mathtt{F}}$ are the average responses for the $\sigma_{\mathtt{T}}$ and $\sigma_{\mathtt{F}}$ data, respectively.
- Let $\{x_{j,k}\}_{k=1}^{m}$ denote the possible values of $x_j, j = 1, \ldots, p$ within the training subset σ (with $m \leq n$ elements).
- Note that, for a fixed j, (6) is a piecewise constant function of ξ, and that its minimal value is attained at some value $x_{j,k}$. As a consequence, to minimize (6) over all j and ξ, it suffices to evaluate (6) for each of the $m \times p$ values $x_{j,k}$ and then take the minimizing pair $(j, x_{j,k})$.

Splitting Rules — the classification setting (1)

- For a classification problem, using the indicator loss and a constant regional prediction function, the aim is to choose a splitting rule that minimizes

$$\frac{1}{n} \sum_{(\mathrm{x},y)\in\sigma_{\mathtt{T}}} \mathbb{1}\{y \neq y_{\mathtt{T}}^*\} + \frac{1}{n} \sum_{(\mathrm{x},y)\in\sigma_{\mathtt{F}}} \mathbb{1}\{y \neq y_{\mathtt{F}}^*\}, \tag{7}$$

  where $y_{\mathtt{T}}^* = g^{\mathtt{T}}(\mathrm{x})$ is the most prevalent class (majority vote) in the data set $\sigma_{\mathtt{T}}$ and $y_{\mathtt{F}}^*$ is the most prevalent class in $\sigma_{\mathtt{F}}$.

- If the feature space is $\mathsf{X} = \mathbb{R}^p$ and the splitting rules are of the form $s(\mathrm{x}) = \mathbb{1}\{x_j \leq \xi\}$, then the optimal splitting rule can be obtained in the same way as described above for the regression case; the only difference is that (6) is replaced with (7).

Splitting Rules — the classification setting (2)

- In fact, we can view the minimization of

$$\frac{1}{n} \sum_{(\mathrm{x},y)\in\sigma_{\mathtt{T}}} \mathbb{1}\{y \neq y^*_{\mathtt{T}}\} + \frac{1}{n} \sum_{(\mathrm{x},y)\in\sigma_{\mathtt{F}}} \mathbb{1}\{y \neq y^*_{\mathtt{F}}\},$$

  as minimizing a weighted average of "impurities" of nodes $\sigma_{\mathtt{T}}$ and $\sigma_{\mathtt{F}}$.
- Namely, for an arbitrary training subset $\sigma \subseteq \tau$, if y^* is the most prevalent label, then

$$\begin{aligned}\frac{1}{|\sigma|} \sum_{(\mathrm{x},y)\in\sigma} \mathbb{1}\{y \neq y^*\} &= 1 - \frac{1}{|\sigma|} \sum_{(\mathrm{x},y)\in\sigma} \mathbb{1}\{y = y^*\} \\ &= 1 - p_{y^*} = 1 - \max_{z\in\{0,\ldots,c-1\}} p_z,\end{aligned}$$

  where p_z is the proportion of data points in σ that have class label z, $z = 0, \ldots, c-1$.

Splitting Rules — the classification setting (3)

- The quantity

$$1 - \max_{z \in \{0,\ldots,c-1\}} p_z$$

measures the diversity of the labels in σ and is called the *misclassification impurity*.

- Consequently,

$$\frac{1}{n} \sum_{(\mathrm{x},y) \in \sigma_{\mathtt{T}}} \mathbb{1}\{y \neq y^*_{\mathtt{T}}\} + \frac{1}{n} \sum_{(\mathrm{x},y) \in \sigma_{\mathtt{F}}} \mathbb{1}\{y \neq y^*_{\mathtt{F}}\},$$

is the weighted sum of the misclassification impurities of $\sigma_{\mathtt{T}}$ and $\sigma_{\mathtt{F}}$, with weights by $|\sigma_{\mathtt{T}}|/n$ and $|\sigma_{\mathtt{F}}|/n$, respectively.

- Note that the misclassification impurity only depends on the label proportions rather than on the individual responses. Instead of using the misclassification impurity to decide if and how to split a data set σ, we can use other impurity measures that only depend on the label proportions.

Splitting Rules — the classification setting (4)

- Two popular choices are the *entropy impurity*:

$$-\sum_{z=0}^{c-1} p_z \log_2(p_z)$$

- and the *Gini impurity*:

$$\frac{1}{2}\left(1-\sum_{z=0}^{c-1} p_z^2\right).$$

- All of these impurities are maximal when the label proportions are equal to $1/c$, so during the construction process we aim to minimize the impurity (in the tree leaves).

Splitting Rules — the classification setting (5)

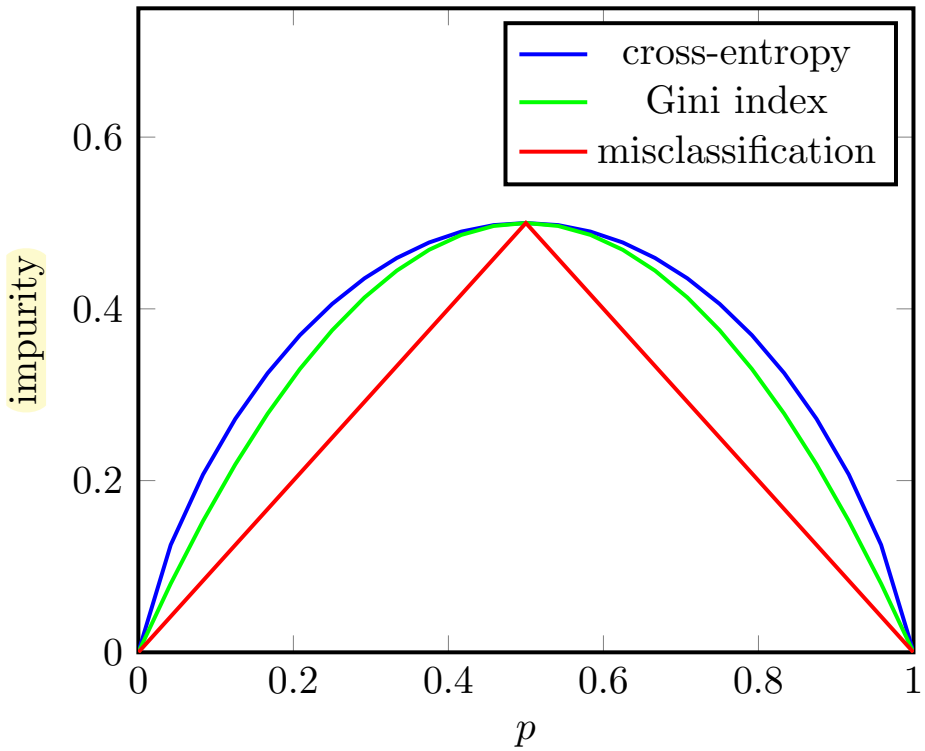


Figure 2: Entropy, Gini, and misclassification impurities for binary classification, with class frequencies $p_1 = p$ and $p_2 = 1 - p$. The entropy impurity was normalized (divided by 2), to ensure that all impurity measures attain the same maximum value of $1/2$ at $p = 1/2$.

The termination criterion

- When building a tree, one can define various types of termination conditions.
- For example, we might stop when the number of data points in the tree node (the size of the input σ set in the tree construction algorithm is less than or equal to some predefined number.
- Or we might choose the maximal depth of the tree in advance.
- Another possibility is to stop when there is no significant advantage, in terms of training loss, to split regions.

Ultimately, the quality of a tree is determined by its predictive performance (generalization risk) and the termination condition should aim to strike a balance between minimizing the approximation error and minimizing the statistical error.

Example — Fixed Tree Depth (1)

To illustrate how the tree depth impacts on the generalization risk, consider Figure 3, which shows the typical behavior of the cross-validation loss as a function of the tree depth.

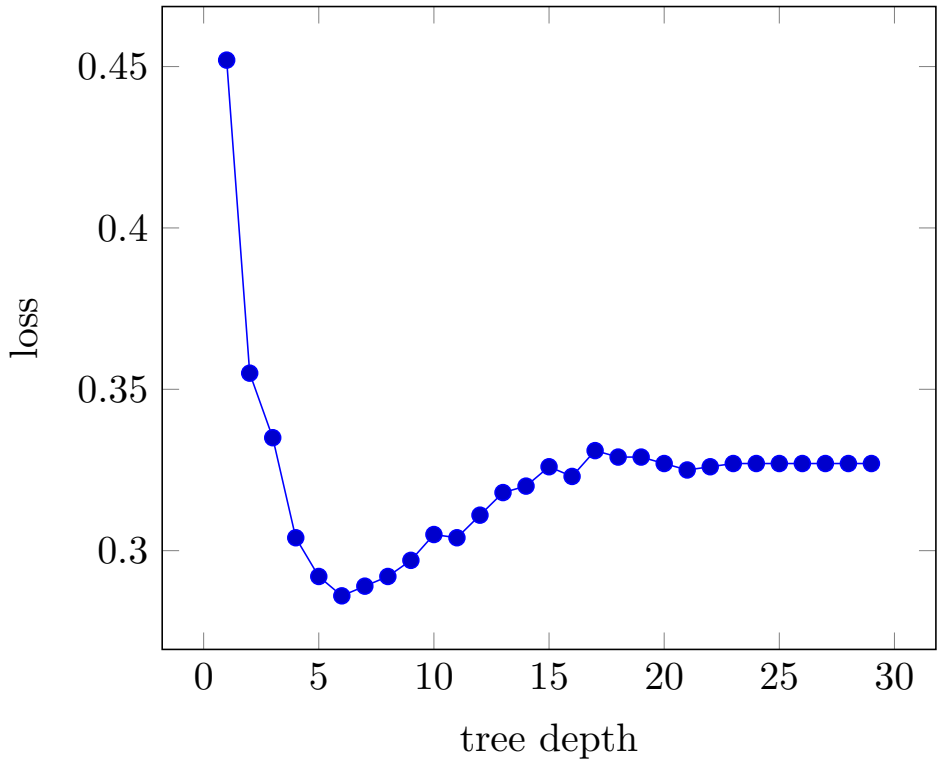


Figure 3: The ten-fold cross-validation loss as a function of the maximal tree depth for a classification problem. The optimal maximal tree depth is here 6.

Example — Fixed Tree Depth (2)

- Recall that the cross-validation loss is an estimate of the expected generalization risk. Complicated (deep) trees tend to overfit the training data by producing many divisions of the feature space.
- As we have seen, this overfitting problem is typical of all learning methods.
- To conclude, increasing the maximal depth does not necessarily result in better performance.

Example — Fixed Tree Depth (3) (TreeDepthCV.py)

```
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import zero_one_loss
import matplotlib.pyplot as plt

def ZeroOneScore(clf, X, y):
    y_pred = clf.predict(X)
    return zero_one_loss(y, y_pred)

# Construct the training set
X, y =  make_blobs(n_samples=5000, n_features=10, centers=3,
                           random_state=10, cluster_std=10)

# construct a decision tree classifier
clf = DecisionTreeClassifier(random_state=0)
```

Example — Fixed Tree Depth (4)

```
# Cross-validation loss as a function of tree depth (1 to 30)
xdepthlist = []
cvlist = []
tree_depth = range(1,30)
for d in tree_depth:
    xdepthlist.append(d)
    clf.max_depth=d
    cv = np.mean(cross_val_score(clf, X, y, cv=10, scoring=ZeroOneScore))
    cvlist.append(cv)

plt.xlabel('tree depth', fontsize=18, color='black')
plt.ylabel('loss', fontsize=18, color='black')
plt.plot(xdepthlist, cvlist,'-*' , linewidth=0.5)
```

The code above relies heavily on sklearn and hides the implementation details. This is very instructive to show how decision trees are actually constructed using the previous theory, we proceed with a very basic implementation.

Basic implementation of a decision tree (1) BasicTree.py

First, we import various packages and define a function to generate the training and test data.

```
import numpy as np
from sklearn.datasets import make_friedman1
from sklearn.model_selection import train_test_split

def makedata():
   n_points = 500 # number of samples

   X, y =  make_friedman1(n_samples=n_points, n_features=5,
                          noise=1.0, random_state=100)
   return train_test_split(X, y, test_size=0.5, random_state=3)
```

Basic implementation of a decision tree (2)

The "main" method calls the *makedata* method, uses the training data to build a regression tree, and then predicts the responses of the test set and reports the mean squared-error loss.

```
def main():
  X_train, X_test, y_train, y_test = makedata()
  maxdepth = 10 # maximum tree depth
  # Create tree root at depth 0
  treeRoot = TNode(0, X_train,y_train)

  # Build the regression tree with maximal depth equal to max_depth
  Construct_Subtree(treeRoot, maxdepth)

  # Predict
  y_hat = np.zeros(len(X_test))
  for i in range(len(X_test)):
     y_hat[i] = Predict(X_test[i],treeRoot)

  MSE = np.mean(np.power(y_hat - y_test,2))
  print("Basic tree: tree loss = ",  MSE)
```

Basic implementation of a decision tree (3)

- The next step is to specify a tree node as a Python class.
- Each node has a number of attributes, including the features and the response data ($\mathtt{X}$ and $\mathtt{y}$) and the depth at which the node is placed in the tree.
- The root node has depth 0.
- Each node w can calculate its contribution to the squared-error training loss $\sum_{i=1}^{n} \mathbb{1}\{\mathrm{x}_i \in R^w\}(y_i - g^w(\mathrm{x}_i))^2$.
- Note that we have omitted the constant $1/n$ term when training the tree, which simply scales the loss.

Basic implementation of a decision tree (4)

```
class TNode:
   def __init__(self, depth, X, y):
      self.depth = depth
      self.X = X   # matrix of features
      self.y = y   # vector of response variables
      # initialize optimal split parameters
      self.j = None
      self.xi = None
      # initialize children to be None
      self.left = None
      self.right = None
      # initialize the regional predictor
      self.g = None

   def CalculateLoss(self):
      if(len(self.y)==0):
         return 0

      return np.sum(np.power(self.y - self.y.mean(),2))
```

Basic implementation of a decision tree — the implementation of the main algorithm (5)

```
def Construct_Subtree(node, max_depth):
    if(node.depth == max_depth or len(node.y) == 1):
        node.g  = node.y.mean()
    else:
        j, xi = CalculateOptimalSplit(node)
        node.j = j
        node.xi = xi
        Xt, yt, Xf, yf = DataSplit(node.X, node.y, j, xi)
        if(len(yt)>0):
            node.left = TNode(node.depth+1,Xt,yt)
            Construct_Subtree(node.left, max_depth)

        if(len(yf)>0):
            node.right = TNode(node.depth+1, Xf,yf)
            Construct_Subtree(node.right, max_depth)
    return node
```

Basic implementation of a decision tree (6)

- This requires an implementation of the *CalculateOptimalSplit* function.
- To start, we implement a function *DataSplit* that splits the data according to $s(x) = \mathbb{1}\{x_j \leq \xi\}$.

```
def DataSplit(X,y,j,xi):
    ids = X[:,j]<=xi
    Xt  = X[ids == True,:]
    Xf  = X[ids == False,:]
    yt  = y[ids == True]
    yf  = y[ids == False]
    return Xt, yt, Xf, yf
```

The *CalculateOptimalSplit* method on the next slide, runs through the possible splitting thresholds ξ from the set $\{x_{j,k}\}$ and finds the optimal split.

Basic implementation of a decision tree (7)

```
def CalculateOptimalSplit(node):
    X = node.X
    y = node.y
    best_var = 0
    best_xi = X[0,best_var]
    best_split_val = node.CalculateLoss()

    m, n  = X.shape

    for j in range(0,n):
        for i in range(0,m):
            xi = X[i,j]
            Xt, yt, Xf, yf = DataSplit(X,y,j,xi)
            tmpt = TNode(0, Xt, yt)
            tmpf = TNode(0, Xf, yf)
            loss_t = tmpt.CalculateLoss()
            loss_f = tmpf.CalculateLoss()
            curr_val =  loss_t + loss_f
            if (curr_val < best_split_val):
                best_split_val = curr_val
                best_var = j
                best_xi = xi
    return best_var,  best_xi
```

Basic implementation of a decision tree (8)

Finally, we implement the recursive method for prediction.

```
def Predict(X,node):
    if(node.right == None and node.left != None):
        return Predict(X,node.left)

    if(node.right != None and node.left == None):
        return Predict(X,node.right)

    if(node.right == None and node.left == None):
        return node.g
    else:
        if(X[node.j] <= node.xi):
            return Predict(X,node.left)
        else:
            return Predict(X,node.right)
```

Basic implementation of a decision tree (9)

In order to test the methods, we run the following function.

```
main()  # run the main program

# compare with sklearn
from sklearn.tree import DecisionTreeRegressor

X_train, X_test, y_train, y_test = makedata() # use the same data
regTree = DecisionTreeRegressor(max_depth = 10, random_state=0)
regTree.fit(X_train,y_train)
y_hat = regTree.predict(X_test)
MSE2 = np.mean(np.power(y_hat - y_test,2))
print("DecisionTreeRegressor: tree loss = ",  MSE2)

Basic tree: tree loss =  9.067077996170276
DecisionTreeRegressor: tree loss =  10.197991295531748
```

Basic tree: tree loss $= 9.067077996170276$

DecisionTreeRegressor: tree loss $= 10.197991295531748$

Additional Considerations: Binary Versus Non-Binary Trees

- While it is possible to split a tree node into more than two groups (multiway splits), it generally produces inferior results compared to the simple binary split.
- The major reason is that multiway splits can lead to too many nodes near the tree root that have only a few data points, thus leaving insufficient data for later splits.
- As multiway splits can be represented as several binary splits, the latter is preferred.

Additional Considerations: Data preprocessing

- Sometimes, it can be beneficial to preprocess the data prior to the tree construction.
- For example, PCA can be used with a view to identify the most important dimensions, which in turn will lead to simpler and possibly more informative splitting rules in the internal nodes.

Additional Considerations: Alternative Splitting Rules (1)

- We restricted our attention to splitting rules of the type

$$s(\mathrm{x}) = \mathbb{1}\{x_j \leq \xi\},$$

  where $j \in \{1, \ldots, p\}$ and $\xi \in \mathbb{R}$.
- These types of rules may not always result in a simple partition of the feature space, as illustrated by the binary data in the Figure on the next slide.
- In this case, the feature space could have been partitioned into just two regions, separated by a straight line.

Additional Considerations: Alternative Splitting Rules (2)

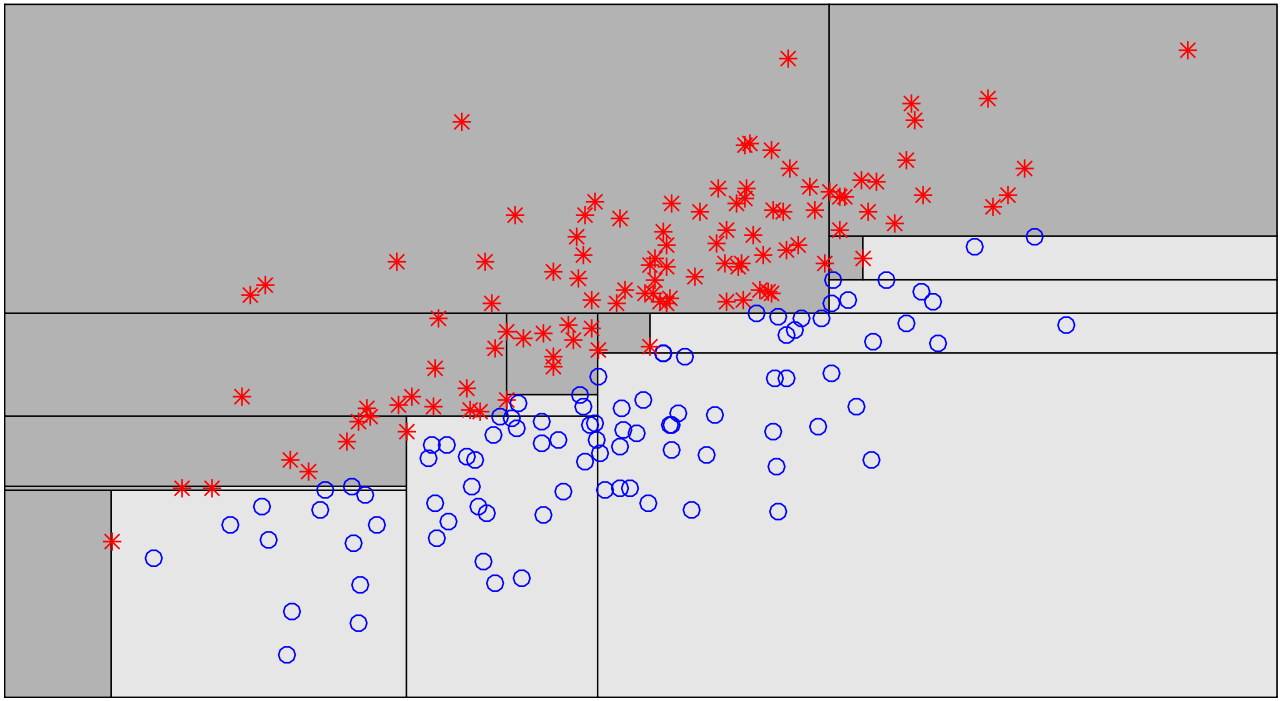


Figure 4: The two groups of points can here be separated by a straight line. Instead, the classification tree divides up the space into many rectangles, leading to an unnecessarily complicated classification procedure.

Additional Considerations: Alternative Splitting Rules (3)

- In this case many classification methods such as linear discriminant analysis will work very well, whereas the classification tree is rather elaborate, dividing the feature set into too many regions.
- An obvious remedy is to use splitting rules of the form

$$s(\mathsf{x}) = \mathbb{1}\{\mathsf{a}^\top \mathsf{x} \leq \xi\}.$$

- In some cases, such as the one just discussed, it may be useful to use a splitting rule that involves several variables, as opposed to a single one.
- The decision regarding the split type clearly depends on the problem domain. For example, for logical (binary) variables our domain knowledge may indicate that a different behavior is expected when both x_i and x_j $(i \neq j)$ are `True`.
- In this case, we will naturally introduce a decision rule of the form:

$$s(\mathsf{x}) = \mathbb{1}\{x_i = \texttt{True} \text{ and } x_j = \texttt{True}\}.$$

Additional Considerations: Categorical Variables

- When an explanatory variable is categorical with labels (levels) say $\{1, \ldots, k\}$, the splitting rule is generally defined via a partition of the label set $\{1, \ldots, k\}$ into two subsets.
- Specifically, let L and R be a partition of $\{1, \ldots, k\}$. Then, the splitting rule is defined via

$$s(\mathrm{x}) = \mathbb{1}\{x_j \in L\}.$$

- For the general supervised learning case, finding the optimal partition in the sense of minimal loss requires one to consider 2^k subsets of $\{1, \ldots, k\}$.
- Consequently, finding a good splitting rule for categorical variables can be challenging when the number of labels p is large.

Additional Considerations: Missing Values (1)

- Missing data is present in many real-life problems.
- Generally, when working with incomplete feature vectors, where one or more values are missing, it is typical to either completely delete the feature vector from the data (which may distort the data) or to impute (guess) its missing values from the available data.
- Tree methods, however, allow an elegant approach for handling missing data.
- When dealing with categorical (factor) features, we can introduce an additional category "missing" for the absent data.
- In the general case, the missing data problem can be handled via *surrogate* splitting rules.

Additional Considerations: Missing Values (2)

- The main idea of surrogate rules is as follows. First, we construct a decision (regression or a classification) tree via the tree construction algorithm.
- During this construction process, the solution of the optimization problem

$$\frac{1}{n}\sum_{i=1}^{n}\mathbb{1}_{\{(\mathrm{x},y)\in\sigma_{\mathtt{T}}\}}\,\mathrm{Loss}(y_i,g^{\mathtt{T}}(\mathrm{x}_i))+\frac{1}{n}\sum_{i=1}^{n}\mathbb{1}_{\{(\mathrm{x},y)\in\sigma_{\mathtt{F}}\}}\,\mathrm{Loss}(y_i,g^{\mathtt{F}}(\mathrm{x}_i)),$$

  is calculated only over the observations that are not missing a particular variable.

Additional Considerations: Missing Values (3)

- Suppose that a tree node v has a splitting rule $s^*(\mathrm{x}) = \mathbb{1}\{x_{j^*} \leq \xi^*\}$ for some $1 \leq j^* \leq p$ and threshold ξ^*.
- For the node v we can introduce a set of alternative splitting rules that resemble the original splitting rule, sometimes called the *primary* splitting rule, using different variables and thresholds.
- Namely, we look for a binary splitting rule $s(\mathrm{x} \mid j, \xi)$, $j \neq j^*$ such that the data split introduced by s will be *similar* to the original data split from s^*.
- The similarity is generally measured via a binary misclassification loss, where the true classes of observations are determined by the primary splitting rule and the surrogate splitting rules serve as classifiers.

Additional Considerations: Missing Values (4)

- Consider, for example, the data in Table 1 and suppose that the primary splitting rule at node v is $\mathbb{1}\{\text{Age} \leq 25\}$.
- That is, the five data points are split such that the left and the right child of v contains two and three data points, respectively.
- Next, the following surrogate splitting rules can be considered:
  1. $\mathbb{1}\{\text{Salary} \leq 1500\}$, and
  2. $\mathbb{1}\{\text{Height} \leq 173\}$.

Table 1: Example data with three variables (Age, Height, and Salary).

Id	Age	Height	Salary
1	20	173	1000
2	25	168	1500
3	38	191	1700
4	49	170	1900
5	62	182	2000

Additional Considerations: Missing Values (5)

- The $\mathbb{1}\{\text{Salary} \leq 1500\}$ surrogate rule completely mimics the primary rule, in the sense that the data splits induced by these rules are identical.
- Namely, both rules partition the data into two sets (by Id) $\{1, 2\}$ and $\{3, 4, 5\}$. On the other hand, the $\mathbb{1}\{\text{Height} \leq 173\}$ rule is less similar to the primary rule, since it causes the different partition $\{1, 2, 4\}$ and $\{3, 5\}$.

Id	Age	Height	Salary
1	20	173	1000
2	25	168	1500
3	38	191	1700
4	49	170	1900
5	62	182	2000

Additional Considerations: Missing Values (6)

- It is up to the user to define the number of surrogate rules for each tree node.
- As soon as these surrogate rules are available, we can use them to handle a new data point, even if the main rule cannot be applied due to a missing value of the primary variable x_{j^*}.
- Specifically, if the observation is missing the primary split variable, we apply the first (best) surrogate rule.
- If the first surrogate variable is also missing, we apply the second best surrogate rule, and so on.

Controlling the Tree Shape (1)

- Eventually, we are interested in getting the right-size tree. Namely, a tree that shows good generalization properties.
- Shallow trees tend to underfit and deep trees tend to overfit the data.
- Basically, a shallow tree does not produce a sufficient number of splits and a deep tree will produce many partitions and thus many leaf nodes.
- If we grow the tree to a sufficient depth, each training sample will occupy a separate leaf and we will observe a zero loss with respect to the training data.
- The above phenomenon is illustrated in Figure 5 (next slide), which presents the cross-validation loss and the training loss as a function of the tree depth.

Controlling the Tree Shape (2)

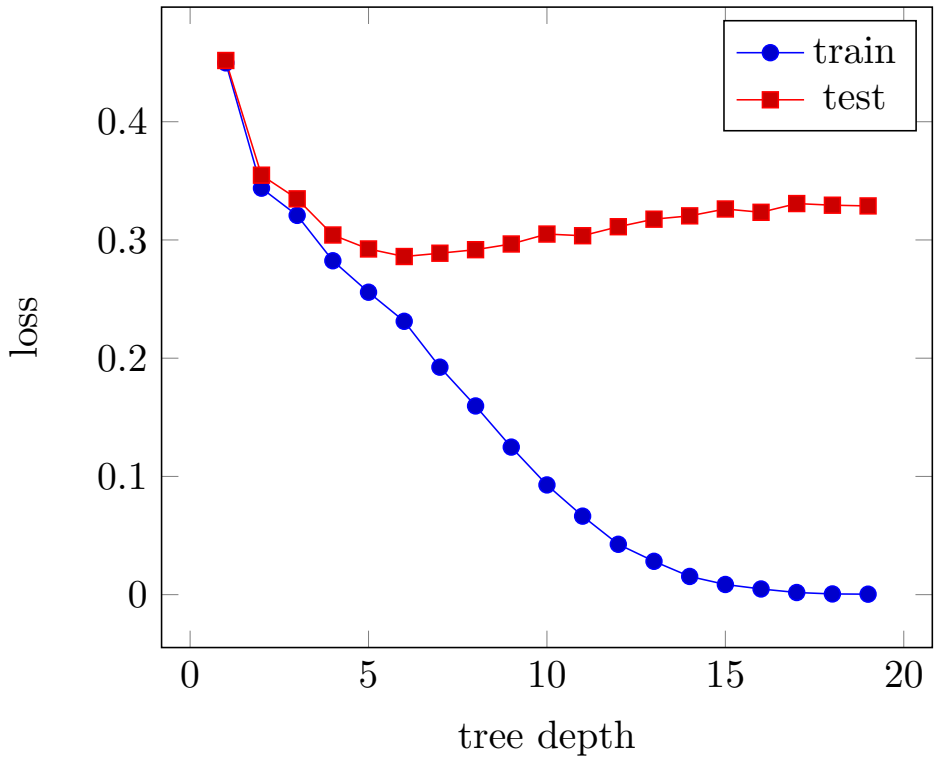


Figure 5: The cross-validation and the training loss as a function of the tree depth for a binary classification problem.

Controlling the Tree Shape (3)

- In order to overcome the under- and the overfitting problem, Breiman examined the possibility of stopping the tree from growing as soon as the decrease in loss due to a split of node v, as expressed in the difference of

$$\frac{1}{n}\sum_{i=1}^{n}\mathbb{1}_{\{(\mathrm{x},y)\in\sigma\}}\,\mathrm{Loss}(y_i, g^{v}(\mathrm{x}_i)).$$

  and

$$\frac{1}{n}\sum_{i=1}^{n}\mathbb{1}_{\{(\mathrm{x},y)\in\sigma_{\mathrm{T}}\}}\,\mathrm{Loss}(y_i, g^{\mathrm{T}}(\mathrm{x}_i)) + \frac{1}{n}\sum_{i=1}^{n}\mathbb{1}_{\{(\mathrm{x},y)\in\sigma_{\mathrm{F}}\}}\,\mathrm{Loss}(y_i, g^{\mathrm{F}}(\mathrm{x}_i)),$$

  is smaller than some predefined parameter $\delta \in \mathbb{R}$.
- Under this setting, the tree construction process will terminate when no leaf node can be split such that the contribution to the training loss after this split is greater than δ.

Controlling the Tree Shape (4)

- However, it was noted that a very small δ leads to an excessive amount of splitting and thus causes overfitting.
- Increasing δ do not work either.
- The problem is that the nature of the proposed rule is *one-step-look-ahead*. To see this, consider a tree node for which the best possible decrease in loss is smaller than δ.
- According to the proposed procedure, this node will not be split further.
- This may, however, be sub-optimal, because it could happen that one of the node's *descendants*, if split, could lead to a major decrease in loss.

Controlling the Tree Shape (5)

- To address these issues, a so-called *pruning* routine can be employed. The idea is as follows.
- We first grow a very deep tree and then prune (remove nodes) it upwards until we reach the root node.
- Consequently, the pruning process causes the number of tree nodes to decrease.
- While the tree is being pruned, the generalization risk gradually decreases up to the point where it starts increasing again, at which point the pruning is stopped.
- This decreasing/increasing behavior is due to the bias–variance tradeoff.
- To start with, let v and v' be tree nodes. We say that v' is a *descendant* of v if there is a path down the tree, which leads from v to v'.
- If such a path exists, we also say that v is an *ancestor* of v'. Consider the tree in Figure 6 (see next slide).

Controlling the Tree Shape (6)

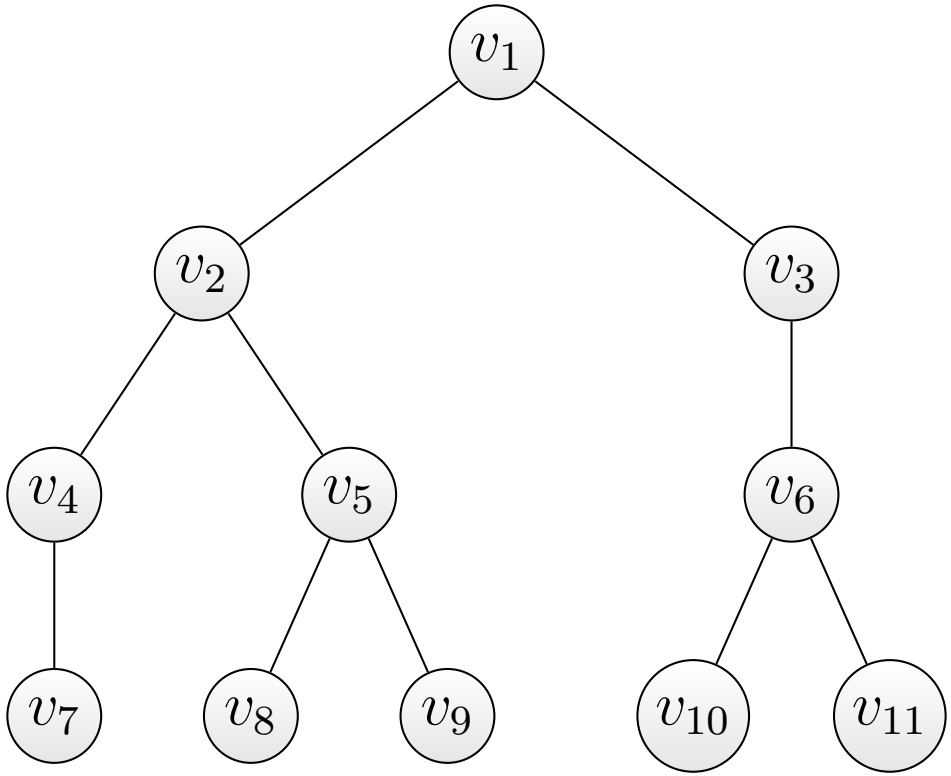


Figure 6: The node v_9 is a descendant of v_2, and v_2 is an ancestor of $\{v_4, v_5, v_7, v_8, v_9\}$, but v_6 is not a descendant of v_2.

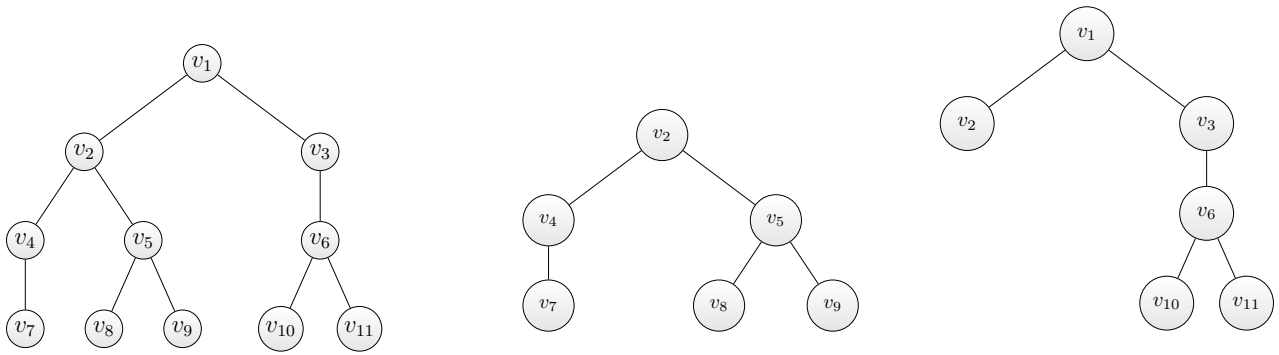
Controlling the Tree Shape (7)

To formally define pruning, we will require the following Definition 1. An example of pruning is demonstrated in Figure 7.

Definition (Branches and Pruning)

1. A tree branch T_v of the tree T is a sub-tree of T rooted at node $v \in T$.
2. The pruning of branch T_v from a tree T is performed via deletion of the entire branch T_v from T except the branch's root node v. The resulting pruned tree is denoted by $T - T_v$.
3. A sub-tree $T - T_v$ is called a pruned sub-tree of T. We indicate this with the notation $T - T_v \prec T$ or $T \succ T - T_v$.

Controlling the Tree Shape (8)



(a) T (b) T_{v_2} (c) $T - T_{v_2}$

Figure 7: The pruned tree $T - T_{v_2}$ in (c) is the result of pruning the T_{v_2} branch in (b) from the original tree T in (a).

A basic decision tree pruning procedure is summarized in Algorithm 2 (see next slide).

Controlling the Tree Shape (9)

Algorithm 2: Decision Tree Pruning

Input: Training set τ.

Output: Sequence of decision trees $T^0 \succ T^1 \succ \cdots$

Build a large decision tree T^0 via the recursive tree construction algorithm. [A possible termination criterion for that algorithm is to have some small predetermined number of data points at each terminal node of T^0.]

$T' \leftarrow T^0$

$k \leftarrow 0$

while T' has more than one node **do**

  $k \leftarrow k + 1$

  Choose $v \in T'$.

  Prune the branch rooted at v from T'.

  $T^k \leftarrow T' - T_v$ and $T' \leftarrow T^k$.

end

return $T^0, T^1, \ldots, T^k$

Controlling the Tree Shape (10)

- Let T^0 be the initial (deep) tree and let T^k be the tree obtained after the k-th pruning operation, for $k = 1, \ldots, K$.
- As soon as the sequence of trees $T^0 \succ T^1 \succ \cdots \succ T^K$ is available, one can choose the best tree of $\{T^k\}_{k=1}^K$ according to the smallest generalization risk.
- Specifically, we can split the data into training and validation sets. In this case, Algorithm 2 is executed using the training set and the generalization risks of $\{T^k\}_{k=1}^K$ are estimated via the validation set.
- While Algorithm 2 and the corresponding best tree selection process look appealing, there is still an important question to consider; namely, how to choose the node v and the corresponding branch T_v in Line 6 of the algorithm.
- In order to overcome this problem, Breiman proposed a method called *cost complexity pruning*, which we discuss next.

Controlling the Tree Shape (11)

- Let $T \prec T^0$ be a tree obtained via pruning of a tree T^0.
- Denote the set of leaf (terminal) nodes of T by $\mathcal{W}$. The number of leaves $|\mathcal{W}|$ is a measure for the complexity of the tree; recall that $|\mathcal{W}|$ is the number of regions $\{\mathcal{R}_w\}$ in the partition of $\mathcal{X}$.
- Corresponding to each tree T is a prediction function g, as in (1).
- In cost-complexity pruning the objective is to find a prediction function g (or, equivalently, tree T) that minimizes the training loss $\ell_\tau(g)$ while taking into account the complexity of the tree.
- The idea is to regularize the training loss, by adding a penalty term for the complexity of the tree. This leads to the following definition.

Controlling the Tree Shape (12)

Definition (Cost-Complexity Measure)

Let $\tau = \{(x_i, y_i)\}_{i=1}^n$ be a data set and $\gamma \geq 0$ be a real number. For a given tree T, the cost-complexity measure $C_\tau(\gamma, T)$ is defined as:

$$C_\tau(\gamma, T) := \frac{1}{n} \sum_{w \in W} \left(\sum_{i=1}^{n} \mathbb{1}\{x_i \in R_w\} \operatorname{Loss}(y_i, g^w(x_i)) \right) + \gamma |W|$$
$$= \ell_\tau(g) + \gamma |W|,$$

where $\ell_\tau(g)$ is the training loss.

Controlling the Tree Shape (13)

$$C_\tau(\gamma, T) := \ell_\tau(g) + \gamma\,|W|,$$

- Small values of γ result in a small penalty for the tree complexity $|W|$, and thus large trees (that fit the entire *training* data well) will minimize the measure $C_\tau(\gamma, T)$. In particular, for $\gamma = 0$, $T = T^0$ will be the minimizer of $C_\tau(\gamma, T)$.
- On the other hand, large values of γ will prefer smaller trees or, more precisely, trees with fewer leaves. For sufficiently large γ, the solution T will collapse to a single (root) node.
- It can be shown that, for every value of γ, there exists a smallest minimizing sub-tree with respect to the cost-complexity measure. In practice, a suitable γ is selected via observing the performance of the learner on the validation set or by cross-validation.

Advantages of Decision Trees (1)

- The tree structure can handle both categorical and numerical features in a natural and straightforward way. Specifically, there is no need to pre-process categorical features, say via the introduction of dummy variables.
- The final tree obtained after the training phase can be compactly stored for the purpose of making predictions for new feature vectors. The prediction process only involves a single tree traversal from the tree root to a leaf.
- The hierarchical nature of decision trees allows for an efficient encoding of the feature's conditional information. Specifically, after an internal split of a feature x_j via the standard splitting rule we will only consider such subsets of data that were constructed based on this split, thus implicitly exploiting the corresponding conditional information from the initial split of x_j.

Advantages of Decision Trees (2)

- The tree structure can be easily understood and interpreted by domain experts with little statistical knowledge, since it is essentially a logical decision flow diagram.
- The sequential decision tree growth procedure in the recursive Algorithm 1, and in particular the fact that the tree has been split using the most important features, provides an implicit step-wise variable elimination procedure. In addition, the partition of the variable space into smaller regions results in simpler prediction problems in these regions.
- Decision trees are invariant under monotone transformations of the data. To see this, consider the (optimal) splitting rule $s(\mathrm{x}) = \mathbb{1}\{x_3 \leq 2\}$, where x_3 is a positive feature. Suppose that x_3 is transformed to $x_3' = x_3^2$. Now, the optimal splitting rule will take the form $s(\mathrm{x}) = \mathbb{1}\{x_3' \leq 4\}$.

Advantages of Decision Trees (3)

- In the classification setting, it is common to report not only the predicted value of a feature vector, but also the respective class probabilities. Decision trees handle this task without any additional effort. Specifically, consider a new feature vector. During the estimation process, we will perform a tree traversal and the point will end up in a certain leaf w. The probability of this feature vector lying in class z can be estimated as the proportion of training points in w that are in class z.
- As each training point is treated equally in the construction of a tree, their structure of the tree will be relatively robust to outliers. In a way, trees exhibit a similar kind of robustness as the sample median does for real-valued data.

Limitations of Decision Trees

- Despite the fact that the decision trees are extremely interpretable, the predictive accuracy is generally inferior to other established statistical learning methods.
- In addition, decision trees, and in particular very deep trees that were not subject to pruning, are heavily reliant on their training set.
- A small change in the training set can result in a dramatic change of the resulting decision tree.
- Their inferior predictive accuracy, however, is a direct consequence of the bias–variance tradeoff. Specifically, a decision tree model generally exhibits a high variance.
- To overcome the above limitations, several promising approaches such as *bagging*, *random forest*, and *boosting* were developed.

Bootstrap Aggregation (1)

- The major idea of the *bootstrap aggregation* or *bagging* method is to combine prediction functions learned from multiple data sets, with a view to improving overall prediction accuracy.
- Bagging is especially beneficial when dealing with predictors that tend to overfit the data, such as in decision trees, where the (unpruned) tree structure is very sensitive to small changes in the training set.
- To start with, consider an idealized setting for a regression tree, where we have access to B iid copies[1] $\mathcal{T}_1, \ldots, \mathcal{T}_B$ of a training set $\mathcal{T}$.
- Then, we can train B separate regression models (B different decision trees) using these sets, giving learners $g_{\mathcal{T}_1}, \ldots, g_{\mathcal{T}_B}$, and take their average:

$$g_{\text{avg}}(\mathbf{x}) = \frac{1}{B} \sum_{b=1}^{B} g_{\mathcal{T}_b}(\mathbf{x}). \tag{8}$$

[1] In this section $\mathcal{T}_k$ means the k-th training set, not a training set of size k.

Bootstrap Aggregation (2)

- By the law of large numbers, as $B \to \infty$, the average prediction function converges to the expected prediction function $g^\dagger := \mathbb{E} g_{\mathcal{T}}$.
- The following result shows that using $g^\dagger$ as a prediction function (if it were known) would result in an expected squared-error generalization risk that is less than or equal to the expected generalization risk for a general prediction function $g_{\mathcal{T}}$.
- It thus suggests that taking an average of prediction functions may lead to a better expected squared-error generalization risk.

Theorem

Expected Squared-Error Generalization Riskbootstrap Let $\mathcal{T}$ be a random training set and let X, Y be a random feature vector and response that are independent of $\mathcal{T}$. Then,

$$\mathbb{E}\left(Y - g_{\mathcal{T}}(\mathsf{X})\right)^2 \geq \mathbb{E}\left(Y - g^\dagger(\mathsf{X})\right)^2.$$

Bootstrap Aggregation (3)

Proof.

We have

$$\mathbb{E}\left[\left(Y - g_{\hat{T}}(X)\right)^2 \middle| X, Y\right] \geq \left(\mathbb{E}[Y \mid X, Y] - \mathbb{E}[g_{\hat{T}}(X) \mid X, Y]\right)^2$$
$$= \left(Y - g^{\dagger}(X)\right)^2,$$

where the inequality follows from $\mathbb{E}U^2 \geq (\mathbb{E}U)^2$ for any (conditional) expectation. Consequently, by the tower property,

$$\mathbb{E}\left(Y - g_{\hat{T}}(X)\right)^2 = \mathbb{E}\left[\mathbb{E}\left[\left(Y - g_{\hat{T}}(X)\right)^2 \mid X, Y\right]\right] \geq \mathbb{E}\left(Y - g^{\dagger}(X)\right)^2.$$

□

Bootstrap Aggregation (4)

- Unfortunately, multiple independent data sets are rarely available. But we can substitute them by bootstrapped ones.
- Specifically, instead of the $\mathcal{T}_1, \ldots, \mathcal{T}_B$ sets, we can obtain random training sets $\mathcal{T}_1^*, \ldots, \mathcal{T}_B^*$ by resampling them from a *single* (fixed) training set τ, and use them to train B separate models.
- By model averaging we obtain the bootstrapped aggregated estimator or *bagged estimator of the form:*

$$g_{\text{bag}}(\mathrm{x}) = \frac{1}{B} \sum_{b=1}^{B} g_{\mathcal{T}_b^*}(\mathrm{x}). \tag{9}$$

The Bootstrap Sampling Algorithm

Algorithm 3: Bootstrap Aggregation Sampling

Input: Training set $\tau = \{(x_i, y_i)\}_{i=1}^{n}$ and resample size B.

Output: Bootstrapped data sets.

for $b = 1$ **to** B **do**
  $\mathcal{T}_b^* \leftarrow \emptyset$
  for $i = 1$ **to** n **do**
    Draw $U \sim \mathrm{U}(0, 1)$;
    $I \leftarrow \lceil nU \rceil$ `// select random index;`
    $\mathcal{T}_b^* \leftarrow \mathcal{T}_b^* \cup \{(x_I, y_I)\}$.
  end
end
return $\mathcal{T}_b^*, b = 1, \ldots, B$.

Bootstrap Aggregation for Classification Problems

- Note that the bagging idea was discussed in the context of regression problems.
- However, the bagging idea can be readily extended to handle classification settings as well.
- For example, g_{bag} can take the majority vote among $\{g_{\mathsf{T}_b^*}\}, b = 1, \ldots, B$; that is, to accept the most frequent class among B predictors.
- While bagging can be applied for any statistical model (such as decision trees, neural networks, linear regression, K-nearest neighbors, and so on), it is most effective for predictors that are sensitive to small changes in the training set.

Where can we use the Bootstrap Aggregation? (1)

- The reason becomes clear when we decompose the expected generalization risk as

$$\mathbb{E}\,\ell(g_{\mathcal{T}}) = \ell^* + \underbrace{\mathbb{E}\,(\mathbb{E}[g_{\mathcal{T}}(\mathsf{X}) \mid \mathsf{X}] - g^*(\mathsf{X}))^2}_{\text{expected squared bias}} + \underbrace{\mathbb{E}[\mathrm{Var}[g_{\mathcal{T}}(\mathsf{X}) \mid \mathsf{X}]]}_{\text{expected variance}}, \quad (10)$$

- Compare this with the same decomposition for the average prediction function g_{bag} in $g_{\mathrm{avg}}(\mathsf{x}) = \frac{1}{B}\sum_{b=1}^{B} g_{\mathcal{T}_b}(\mathsf{x})$.
- As $\mathbb{E}g_{\mathrm{bag}}(\mathsf{x}) = \mathbb{E}g_{\mathcal{T}}(\mathsf{x})$, we see that any possible improvement in the generalization risk must be due to the expected variance term.
- Averaging and bagging are thus only useful for predictors with a large expected variance, relative to the other two terms.

Where can we use the Bootstrap Aggregation? (2)

- Examples of such "unstable" predictors include decision trees, neural networks, and subset selection in linear regression.
- On the other hand, "stable" predictors are insensitive to small data changes, an example being the K-nearest neighbors method.
- Note that for independent training sets $\mathcal{T}_1, \ldots, \mathcal{T}_B$ a reduction of the variance by a factor B is achieved:

$$\operatorname{Var} g_{\text{bag}}(\mathrm{x}) = B^{-1} \operatorname{Var} g_{\mathcal{T}}(\mathrm{x}).$$

- Again, it depends on the squared bias and irreducible loss how significant this reduction is for the generalization risk.

Limitations of Bagging

- It is important to remember that g_{bag} is not exactly equal to g_{avg}, which in turn is not exactly $g^{\dagger}$.
- Specifically, g_{bag} is constructed from the bootstrap approximation of the sampling pdf f.
- As a consequence, for stable predictors, it can happen that g_{bag} will perform worse than $g_{\mathcal{T}}$.
- In addition to the deterioration of the bagging performance for stable procedures, it can also happen that $g_{\mathcal{T}}$ has already achieved a near optimal predictive accuracy given the available training data.
- In this case, bagging will not introduce a significant improvement.

Out-of-Bag Loss Estimation (1)

- The bagging process provides an opportunity to estimate the generalization risk of the bagged model without an additional test set.
- Specifically, recall that we obtain the $\mathcal{T}_1^*, \ldots, \mathcal{T}_B^*$ sets from a single training set τ by sampling via the bootstrap sampling algorithm, and use them to train B separate models.
- It can be shown that, for large sample sizes, on average about a third (more precisely, a fraction $\mathrm{e}^{-1} \approx 0.37$) of the original sample points are not included in bootstrapped set $\mathcal{T}_b^*$ for $1 \leq b \leq B$.
- Therefore, these samples can be used for the loss estimation. These samples are called *out-of-bag* (OOB) observations.

Out-of-Bag Loss Estimation (2)

- For each sample from the original data set, we calculate the OOB loss using predictors that were trained without this particular sample.
- Tibshirani observe that the OOB loss is almost identical to the n-fold cross-validation loss.
- In addition, the OOB loss can be used to determine the number of trees required.
- Specifically, we can train predictors until the OOB loss stops changing. Namely, decision trees are added until the OOB loss stabilizes.

The Out-of-Bag Algorithm

Algorithm 4: Out-of-Bag Loss Estimation

Input: The original data set $\tau = \{(\mathrm{x}_1, y_1), \ldots, (\mathrm{x}_n, y_n)\}$, the bootstrapped data sets $\{\mathcal{T}_1^*, \ldots, \mathcal{T}_B^*\}$, and the trained predictors $\left\{g_{\mathcal{T}_1^*}, \ldots, g_{\mathcal{T}_B^*}\right\}$.

Output: Out-of-bag loss for the averaged model.

for $i = 1$ **to** n **do**
  $C_i \leftarrow \emptyset$; // Indices of predictors not depending on (x_i, y_i)
  for $b = 1$ **to** B **do**
    if $(\mathrm{x}_i, y_i) \notin \mathcal{T}_b^*$ **then** $C_i \leftarrow C_i \cup \{b\}$;
  end
  $Y_i' \leftarrow |C_i|^{-1} \sum_{b \in C_i} g_{\mathcal{T}_b^*}(\mathrm{x}_i)$
  $L_i \leftarrow \mathrm{Loss}\,(y_i, Y_i')$
end
$L_{\mathrm{OOB}} \leftarrow \frac{1}{n} \sum_{i=1}^{n} L_i$
return L_{OOB}.

Bagging for a Regression Tree Example (1) BaggingExample.py

Consider a basic bagging example for a regression tree, in which we compare the decision tree estimator with the corresponding bagged estimator. We use the R^2 metric (coefficient of determination) for comparison.

```
import numpy as np
from sklearn.datasets import make_friedman1
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score
# create regression problem
n_points = 1000 # points
x, y =  make_friedman1(n_samples=n_points, n_features=15,
                       noise=1.0, random_state=100)

# split to train/test set
x_train, x_test, y_train, y_test = \
        train_test_split(x, y, test_size=0.33, random_state=100)
```

Bagging for a Regression Tree Example (2)

Simple regression tree (single tree) code.

```
# training
regTree = DecisionTreeRegressor(random_state=100)
regTree.fit(x_train,y_train)
# test
yhat = regTree.predict(x_test)

print("DecisionTreeRegressor R^2 score = ",r2_score(y_test, yhat))
```

Bagging for a Regression Tree Example (3)

Constructing bagged estimator with 500 bootstrapped datasets.

```
# Bagging construction
n_estimators=500
bag = np.empty((n_estimators), dtype=object)
bootstrap_ds_arr = np.empty((n_estimators), dtype=object)
for i in range(n_estimators):
    # sample bootstrapped data set
    ids = np.random.choice(range(0,len(x_test)),size=len(x_test),
                           replace=True)
    x_boot = x_train[ids]
    y_boot = y_train[ids]
    bootstrap_ds_arr[i] = np.unique(ids)

    bag[i] = DecisionTreeRegressor()
    bag[i].fit(x_boot,y_boot)
```

Bagging for a Regression Tree Example (4)

Constructing prediction and OOB estimator.

```
# bagging prediction
yhatbag = np.zeros(len(y_test))
for i in range(n_estimators):
    yhatbag = yhatbag + bag[i].predict(x_test)

yhatbag = yhatbag/n_estimators

# out of bag loss estimation
oob_pred_arr = np.zeros(len(x_train))
for i in range(len(x_train)):
    x = x_train[i].reshape(1, -1)
    C = []
    for b in range(n_estimators):
        if(np.isin(i, bootstrap_ds_arr[b])==False):
            C.append(b)
    for pred in  bag[C]:
        oob_pred_arr[i] = oob_pred_arr[i] + (pred.predict(x)/len(C))
}
L_oob = r2_score(y_train, oob_pred_arr)
```

Bagging for a Regression Tree Example (5)

```
DecisionTreeRegressor R^2 score =  0.575438224929718
Bagging R^2 score =  0.7612121189201985
Bagging OOB R^2 score =  0.7758253149069059
```

The results are as follows.

- The decision tree bagging improves the test-set R^2 score by about 32% (from 0.575 to 0.761).
- Moreover, the OOB score (0.776) is very close to the true generalization risk (0.761) of the bagged estimator.

Random Forests (1)

- We discussed the intuition behind the prediction averaging procedure.
- Specifically, for some feature vector x let $Z_b = g_{\mathcal{T}_b}(\mathrm{x}), b = 1, 2, \ldots, B$ be iid prediction values, obtained from independent training sets $\mathcal{T}_1, \ldots, \mathcal{T}_B$.
- Suppose that $\operatorname{Var} Z_b = \sigma^2$ for all $b = 1, \ldots, B$. Then the variance of the average prediction value $\overline{Z}_B$ is equal to σ^2/B.
- However, if bootstrapped data sets $\{\mathcal{T}_b^*\}$ are used instead, the corresponding random variables $\{Z_b\}$ will be *correlated*.
- In particular, $Z_b = g_{\mathcal{T}_b^*}(\mathrm{x})$ for $b = 1, \ldots, B$ are identically distributed (but not independent) with some positive pairwise correlation ρ. It then holds that

$$\operatorname{Var} \overline{Z}_B = \rho\, \sigma^2 + \sigma^2 \frac{(1-\rho)}{B}. \tag{11}$$

Random Forests (2)

- While the second term of

$$\operatorname{Var} \overline{Z}_B = \rho\, \sigma^2 + \sigma^2 \frac{(1-\rho)}{B}$$

  goes to zero as the number of observation B increases, the first term remains constant.
- This issue is particularly relevant for bagging with decision trees.
- For example, consider a situation in which there exists a feature that provides a very good split of the data.
- Such a feature will be selected and split for every $\{g_{\mathsf{T}_b^*}\}_{b=1}^B$ at the root level and we will consequently end up with highly correlated predictions.
- In such a situation, prediction averaging will not introduce the desired improvement in the performance of the bagged predictor.

Random Forests (3)

- The major idea of random forests is to perform bagging in combination with a "decorrelation" of the trees by including only a subset of features during the tree construction.
- For each bootstrapped training set T_b^* we build a decision tree using a randomly selected subset of $m \leq p$ features for the splitting rules.
- This simple but powerful idea will decorrelate the trees, since strong predictors will have a smaller chance to be considered at the root levels.
- Consequentially, we can expect to improve the predictive performance of the bagged estimator. The resulting predictor (random forest) construction is summarized in the following Algorithm.

Random Forests Construction Algorithm

Algorithm 5: Random Forest Construction

Input: Training set $\tau = \{(x_i, y_i)\}_{i=1}^{n}$, the number of trees in the forest B, and the number $m \leq p$ of features to be included, where p is the total number of features in x.

Output: Ensemble of trees.

1 Generate bootstrapped training sets $\{\mathcal{T}_1^*, \ldots, \mathcal{T}_B^*\}$ via Algorithm 3.
2 **for** $b = 1$ **to** B **do**
3 | Randomly select m out of p features, without replacement.
4 | Using only these features, train a decision tree $g_{\mathcal{T}_b^*}$ via the tree construction algorithm.
5 **end**
6 **return** $\{g_{\mathcal{T}_b^*}\}_{b=1}^{B}$.

Random Forests Example: BaggingExampleRF.py

- For regression problems, the output of Algorithm 5 is combined to yield the random forest prediction function:

$$g_{\mathrm{RF}}(\mathrm{x}) = \frac{1}{B}\sum_{b=1}^{B} g_{\mathsf{T}_b^*}(\mathrm{x}).$$

- In the classification setting, we take instead the majority vote from the $\{g_{\mathsf{T}_b^*}\}$.

Generally, a random forest will outperform bagging estimators. We next continue the example in which we compared the decision tree estimator with the corresponding bagged estimator. It can be seen that the random forest's R^2 score is outperforming that of the bagged estimator. Specifically, we obtained:

```
RF R^2 score =  0.8106589580845707
RF OOB R^2 score =  0.8260541058404149
```

```
from sklearn.datasets import make_friedman1
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score
from sklearn.ensemble import RandomForestRegressor

# create regression problem
n_points = 1000 # points
x, y =  make_friedman1(n_samples=n_points, n_features=15,
                       noise=1.0, random_state=100)
# split to train/test set
x_train, x_test, y_train, y_test = \
        train_test_split(x, y, test_size=0.33, random_state=100)
rf = RandomForestRegressor(n_estimators=500, oob_score = True, max_features=8,
                      random_state=100)
rf.fit(x_train,y_train)
yhatrf = rf.predict(x_test)

print("RF R^2 score = ", r2_score(y_test, yhatrf),
      "\nRF OOB R^2 score = ", rf.oob_score_)
```

Random Forests — The Optimal Number of Subset Features m

- The default values for m are $\lfloor p/3 \rfloor$ and $\lfloor \sqrt{p} \rfloor$ for regression and classification setting, respectively.
- However, the standard practice is to treat m as a hyperparameter that requires tuning, depending on the specific problem at hand.

Note that the procedure of bagging decision trees is a special case of a random forest construction. Consequently, the OOB loss is readily available for random forests.

Random Forests — inference

- While the advantage of bagging in the sense of enhanced accuracy is clear, we should also consider its negative aspects and, in particular, the loss of interpretability.
- Specifically a random forest consists of many trees, thus making the prediction process both hard to visualize and interpret.
- For example, given a random forest, it is not easy to determine a subset of features that are essential for accurate prediction.
- The feature importance measure intends to address this issue. The idea is as follows.

Random Forests — feature importance (1)

- Each *internal* node of a decision tree induces a certain decrease in the training loss.
- Let us denote this decrease in the training loss by $\Delta_{\text{Loss}}(v)$, where v is not a leaf node of T.
- In addition, recall that for splitting rules of the type $\mathbb{1}\{x_j \leq \xi\}$ $(1 \leq j \leq p)$, each node v is associated with a feature x_j that determines the split.
- Using the above definitions, we can define the *feature importance* of x_j as

$$\mathcal{I}_T(x_j) = \sum_{v \text{ internal} \in T} \Delta_{\text{Loss}}(v)\ \mathbb{1}\{x_j \text{ is associated with } v\}, \quad 1 \leq j \leq p.$$

Random Forests — feature importance (2)

- While

$$\mathcal{I}_T(x_j) = \sum_{v \text{ internal} \in T} \Delta_{\text{Loss}}(v)\, \mathbb{1}\{x_j \text{ is associated with } v\}, \quad 1 \le j \le p.$$

  is defined for a single tree, it can be readily extended to random forests.

- Specifically, the feature importance in that case will be averaged over all trees of the forest; that is, for a forest consisting of B trees $\{T_1, \ldots, T_B\}$, the feature importance measure is:

$$\mathcal{I}_{\text{RF}}(x_j) = \frac{1}{B} \sum_{b=1}^{B} \mathcal{I}_{T_b}(x_j), \quad 1 \le j \le p.$$

Random Forests — feature importance example (1) VarImportance.py

- We consider a classification problem with 15 features.
- The data is specifically designed to contain only 5 informative features out of 15.
- In the code below, we apply the random forest procedure and calculate the corresponding feature importance measures.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt, pylab

n_points = 1000 # create regression data with 1000 data points
x, y =  make_classification(n_samples=n_points, n_features=15,
  n_informative=5, n_redundant=0, n_repeated=0, random_state=100,
                              shuffle=False)
```

Random Forests — feature importance example (2)

```
rf = RandomForestClassifier(n_estimators=200, max_features="log2")
rf.fit(x,y)

importances = rf.feature_importances_
indices = np.argsort(importances)[::-1]

for f in range(15):
    print("Feature %d (%f)" % (indices[f]+1, importances[indices[f]]))

std = np.std([rf.feature_importances_ for tree in rf.estimators_],
             axis=0)
f = plt.figure()
plt.bar(range(x.shape[1]), importances[indices],
       color="b", yerr=std[indices], align="center")
plt.xticks(range(x.shape[1]), indices+1)
plt.xlim([-1, x.shape[1]])
pylab.xlabel("feature index")
pylab.ylabel("importance")
plt.show()
```

Random Forests — feature importance example

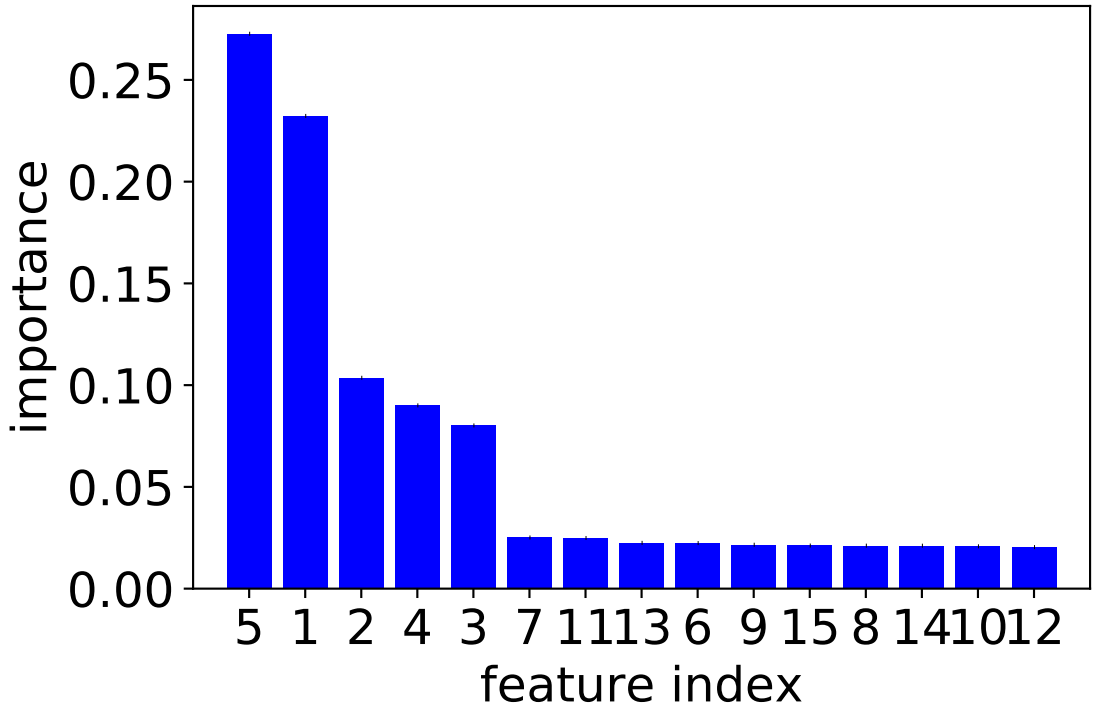


Figure 8: Importance measure for the 15-feature data set with only 5 informative features x_1, x_2, x_3, x_4, and x_5. Clearly, it is hard to visualize and understand the prediction process based on 200 trees. However, Figure 8 shows that the features x_1, x_2, x_3, x_4, and x_5 were correctly identified as being important.

Boosting (1)

- Boosting is a powerful idea that aims to improve the accuracy of any learning algorithm, especially when involving *weak learners* — simple prediction functions that exhibit performance slightly better than random guessing. Shallow decision trees typically yield weak learners.
- Originally, boosting was developed for binary classification tasks, but it can be readily extended to handle general classification and regression problems.
- The boosting approach has some similarity with the bagging method in the sense that boosting uses an ensemble of prediction functions. Despite this similarity, there exists a fundamental difference between these methods.
- Specifically, while bagging involves the fitting of prediction functions to bootstrapped data, the predicting functions in boosting are learned *sequentially*.
- That is, each learner uses information from previous learners.

Boosting (2)

- The idea is to start with a simple model (weak learner) g_0 for the data $\tau = \{(\mathrm{x}_i, y_i)\}_{i=1}^n$ and then to improve or "boost" this learner to a learner

$$g_1 := g_0 + h_1.$$

- Here, the function h_1 is found by minimizing the training loss for $g_0 + h_1$ over all functions h in some class of functions H.
- For example, H could be the set of prediction functions that can be obtained via a decision tree of maximal depth 2.

Given a loss function Loss, the function h_1 is thus obtained as the solution to the optimization problem

$$h_1 = \underset{h \in \mathrm{H}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathrm{Loss}\left(y_i, g_0(\mathrm{x}_i) + h\left(\mathrm{x}_i\right)\right). \tag{12}$$

Boosting (3)

- This process can be repeated for g_1 to obtain $g_2 = g_1 + h_2$, and so on, yielding the boosted prediction function

$$g_B(\mathrm{x}) = g_0 + \sum_{b=1}^{B} h_b(\mathrm{x}). \tag{13}$$

- Instead of using the updating step

$$g_b = g_{b-1} + h_b,$$

one prefers to use the smooth updating step

$$g_b = g_{b-1} + \gamma\, h_b,$$

for some suitably chosen step-size parameter γ. As we shall see shortly, this helps reduce overfitting.

Boosting (4)

- Boosting can be used for regression and classification problems.
- We start with a simple regression setting, using the squared-error loss; thus, $\text{Loss}(y, \hat{y}) = (y - \hat{y})^2$.
- In this case, it is common to start with $g_0(\mathrm{x}) = n^{-1} \sum_{i=1}^{n} y_i$, and each h_b for $b = 1, \ldots, B$ is chosen as a learner for the data set τ_b of residuals corresponding to g_{b-1}.
- That is, $\tau_b := \left\{ \left(\mathrm{x}_i, e_i^{(b)} \right) \right\}_{i=1}^{n}$, with

$$e_i^{(b)} := y_i - g_{b-1}(\mathrm{x}_i). \tag{14}$$

- This leads to the following boosting procedure for regression with squared-error loss.

Boosting — Construction Algorithm

Algorithm 6: Regression Boosting with Squared-Error Loss

Input: Training set $\tau = \{(\mathrm{x}_i, y_i)\}_{i=1}^{n}$, the number of boosting rounds B, and a shrinkage step-size parameter γ.

Output: Boosted prediction function.

1 Set $g_0(\mathrm{x}) \leftarrow n^{-1} \sum_{i=1}^{n} y_i$.

2 **for** $b = 1$ **to** B **do**

3 Set $e_i^{(b)} \leftarrow y_i - g_{b-1}(\mathrm{x}_i)$ for $i = 1, \ldots, n$, and let $\tau_b \leftarrow \left\{ \left(\mathrm{x}_i, e_i^{(b)} \right) \right\}_{i=1}^{n}$.

4 Fit a prediction function h_b on the training data τ_b.

5 Set $g_b(\mathrm{x}) \leftarrow g_{b-1}(\mathrm{x}) + \gamma\, h_b(\mathrm{x})$.

6 **end**

7 **return** g_B.

Boosting step size (1)

- The *step-size parameter* γ introduced in Algorithm 6 controls the speed of the fitting process.
- Specifically, for small values of γ, boosting takes smaller steps towards the training loss minimization.
- The step-size γ is of great practical importance, since it helps the boosting algorithm to avoid overfitting.
- This phenomenon is demonstrated in the Figure on the next slide.

Boosting step size (2)

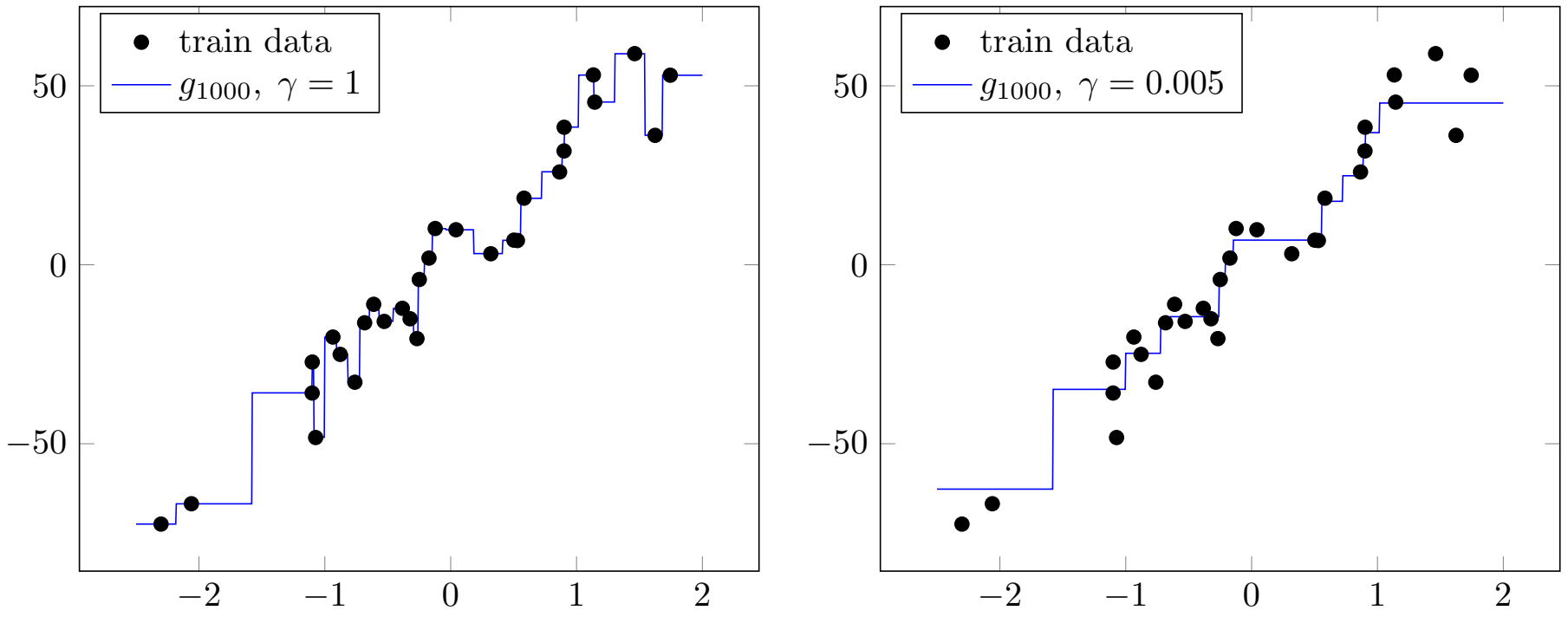


Figure 9: The left and the right panels show the fitted boosting regression model g_{1000} with $\gamma = 1.0$ and $\gamma = 0.005$, respectively. Note the overfitting on the left.

A very basic implementation of 1 which reproduces Figure 9 is provided below.

Boosting — basic example (1) RegressionBoosting.py

```
# create data set
x,y = make_regression(n_samples=sz, n_features=1, n_informative=1,noise=10.0)

# boosting algorithm
BoostingRounds = 1000
alphas = [1, 0.005]

for alpha in alphas:
    g_0, g_boost = TrainBoost(alpha,BoostingRounds,x,y)
    yhat = Predict(g_0, g_boost, alpha, x)

    # plot
    tmpX =  np.reshape(np.linspace(-2.5,2,1000),(1000,1))
    yhatX = Predict(g_0, g_boost, alpha, tmpX)
    f = plt.figure()
    plt.plot(x,y,'*')
    plt.plot(tmpX,yhatX)
    plt.show()
```

Boosting — basic example (2)

```
def TrainBoost(alpha,BoostingRounds,x,y):
    g_0 = np.mean(y)
    residuals  = y-alpha*g_0
    # list of basic regressor
    g_boost = []

    for i in range(BoostingRounds):
        h_i = DecisionTreeRegressor(max_depth=1)
        h_i.fit(x,residuals)
        residuals = residuals  - alpha*h_i.predict(x)
        g_boost.append(h_i)
    return g_0, g_boost

def Predict(g_0, g_boost,alpha, x):
    yhat = alpha*g_0*np.ones(len(x))
    for j in range(len(g_boost)):
        yhat = yhat+alpha*g_boost[j].predict(x)

    return yhat
```

Gradient Boosting (1)

- The parameter γ can be viewed as a step size made in the direction of the negative gradient of the squared-error training loss.
- To see this, note that the negative gradient

$$-\left.\frac{\partial \operatorname{Loss}(y_i, z)}{\partial z}\right|_{z=g_{b-1}(\mathrm{x}_i)} = -\left.\frac{\partial (y_i - z)^2}{\partial z}\right|_{z=g_{b-1}(\mathrm{x}_i)} = 2(y_i - g_{b-1}(\mathrm{x}_i))$$

is two times the residual $e_i^{(b)}$ given in

$$e_i^{(b)} := y_i - g_{b-1}(\mathrm{x}_i).$$

This was used in the Boosting Algorithm to fit the prediction function h_b.

Gradient Boosting (2)

- In fact, one of the major advances in the theory of boosting was the recognition that one can use a similar gradient descent method for any differentiable loss function.
- The resulting algorithm is called *gradient boosting*.
- The general gradient boosting algorithm is summarized on the next slide.
- The main idea is to mimic a gradient descent algorithm in the following sense. At each stage of the boosting procedure, we calculate a negative gradient on n training points $x_1, \ldots, x_n$ (Lines 3–4).
- Then, we fit a simple model (such as a shallow decision tree) to approximate the gradient (Line 6) for any feature x.
- Finally, similar to the gradient descent method, we make a γ-sized step in the direction of the negative gradient (Line 7).

Algorithm 7: Gradient Boosting

Input: Training set $\tau = \{(x_i, y_i)\}_{i=1}^n$, the number of boosting rounds B, a differentiable loss function $\mathrm{Loss}(y, \hat{y})$, and a gradient step-size parameter γ.

Output: Gradient boosted prediction function.

Set $g_0(x) \leftarrow 0$.

for $b = 1$ **to** B **do**

  for $i = 1$ **to** n **do**

    Evaluate the negative gradient of the loss at (x_i, y_i) via

$$r_i^{(b)} \leftarrow - \left. \frac{\partial \, \mathrm{Loss}\,(y_i, z)}{\partial \, z} \right|_{z = g_{b-1}(x_i)} \qquad i = 1, \ldots, n.$$

  end

  Approximate the negative gradient by solving

$$h_b = \underset{h \in \mathsf{H}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=0}^{n} \left(r_i^{(b)} - [g_{b-1}(x_i) + h\,(x_i)] \right)^2. \tag{15}$$

  Set $g_b(x) \leftarrow g_{b-1}(x) + \gamma \, h_b(x)$.

end

return g_B

Gradient Boosting Example GradientBoostingRegression.py

- We continue with the basic bagging and random forest examples for a regression tree, where we compared the standard decision tree estimator with the corresponding bagging and random forest estimators.
- Now, we use the gradient boosting estimator as implemented in `sklearn`.
- We use $\gamma = 0.1$ and perform $B = 100$ boosting rounds.
- As a prediction function h_b for $b = 1, \ldots, B$ we use small regression trees of depth at most 3. Note that such individual trees do not usually give good performance; that is, they are weak prediction functions.
- We can see that the resulting boosting prediction function gives the R^2 score equal to 0.899, which is better than R^2 scores of simple decision tree (0.5754), the bagged tree (0.761), and the random forest (0.8106).

```
import numpy as np
from sklearn.datasets import make_friedman1
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score
# create regression problem
n_points = 1000 # points
x, y =  make_friedman1(n_samples=n_points, n_features=15,
                       noise=1.0, random_state=100)
# split to train/test set
x_train, x_test, y_train, y_test = \
        train_test_split(x, y, test_size=0.33, random_state=100)

# boosting sklearn
from sklearn.ensemble import GradientBoostingRegressor
breg = GradientBoostingRegressor(learning_rate=0.1,
            n_estimators=100, max_depth =3, random_state=100)
breg.fit(x_train,y_train)
yhat = breg.predict(x_test)
print("Gradient Boosting R^2 score = ",r2_score(y_test, yhat))
-----------------------------------------------------------
>>Gradient Boosting R^2 score =  0.8993055635639531
```