Module 9: Tree-Based Methods

Content based on ISLR Chapter 8

Module 8 Outline

- Decision Trees
- Forests
- Gradient Boosting

Decision Trees

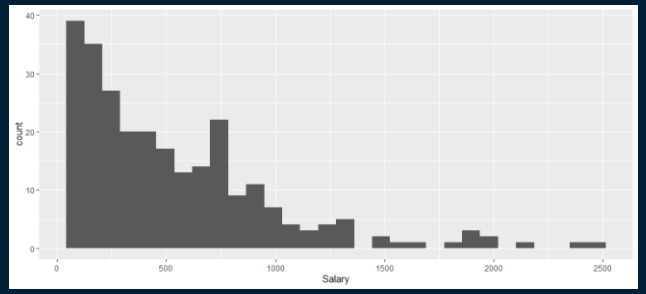
Introduction

- Classic modeling technique that goes back to the 1980s
- Fundamentally different from most other techniques which involve fitting some sort of function to the data
  - Instead, it involves successively partitioning the data according to some sort of “purity” criteria
  - Trying to break up the data space so that the components of the data are a similar as possible
- Can be used for both continuous and categorical targets
  - Commonly referred to as Classification and Regression Trees (CART)
- Approach is to partition the predictor space into several “regions”
  - For classification trees, final prediction is the mode of the training observations in the region
  - For regression trees, final prediction is the mean of the training observations in the region

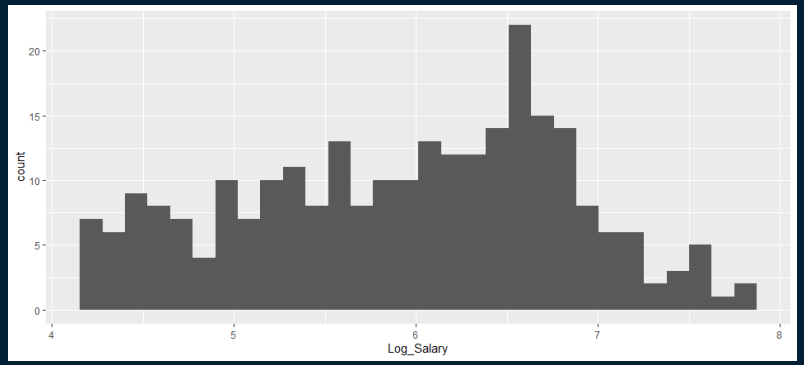
Regression Tree Example – Baseball Example

- We wish to use a regression tree to model a baseball player's salary based on the number of years played in the major league and the number of hits during the previous year.

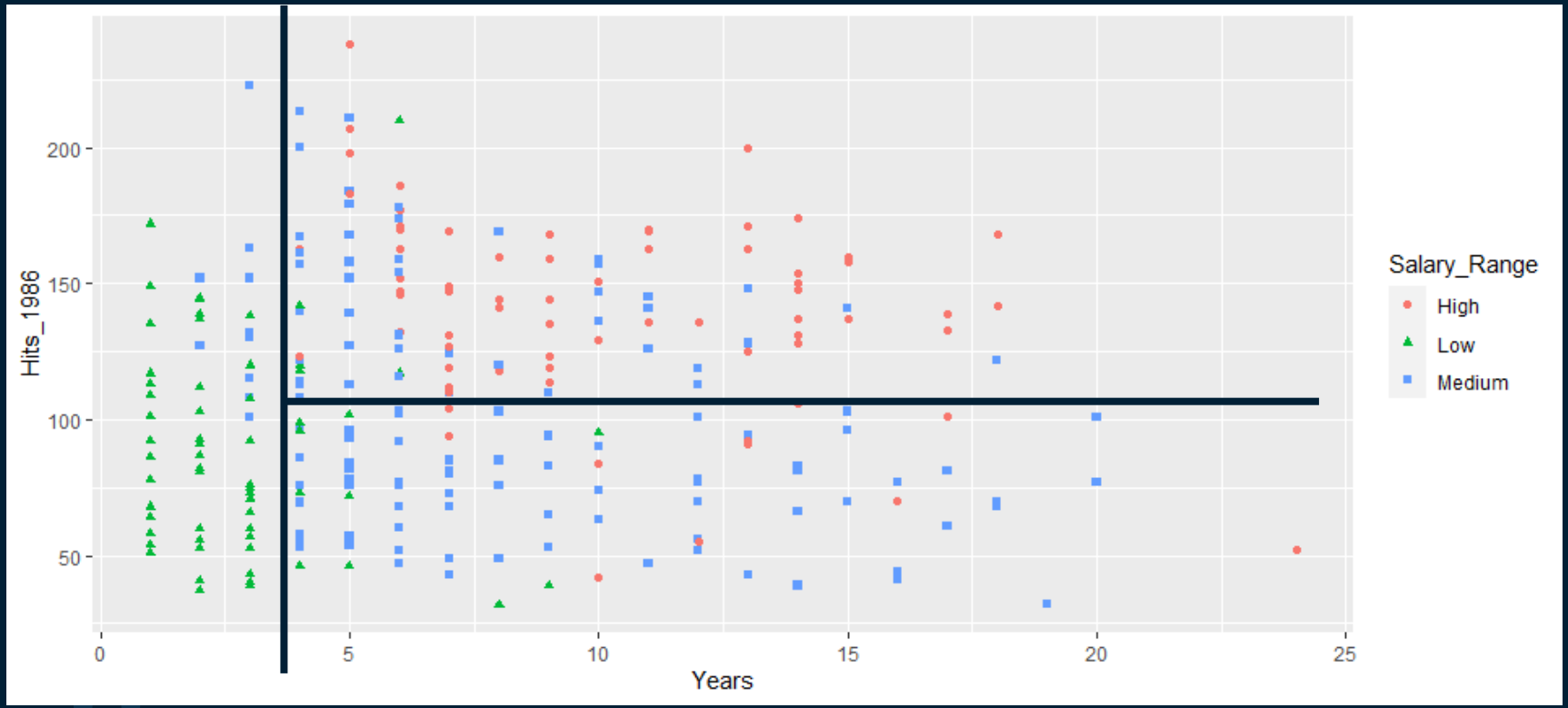
Normality is an assumption for many statistical methods. Salary appears to be heavily right-skewed. Salary



Taking the log compresses the scale of a right-skewed distribution making it approximately normal so we use this in our model.

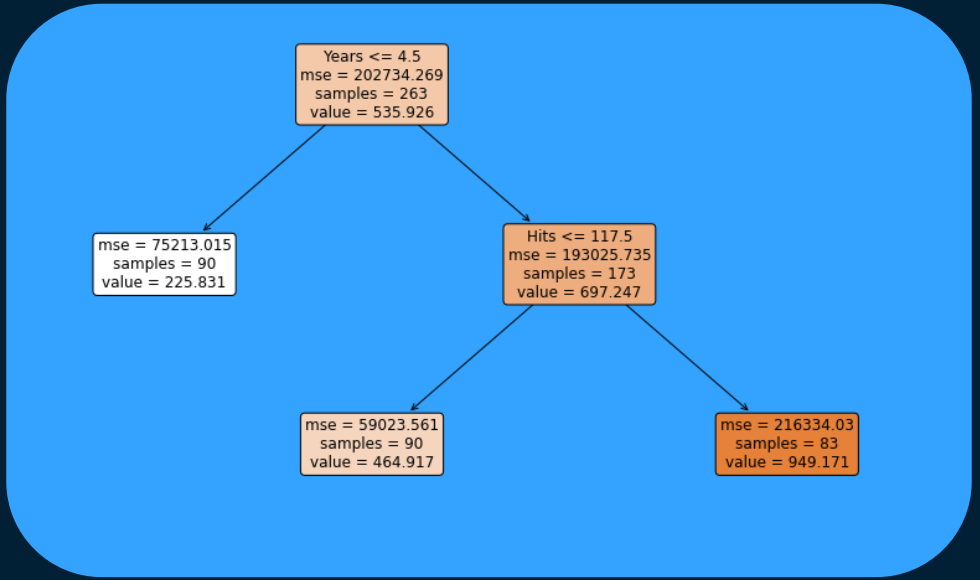


Partitioned Predictor Space



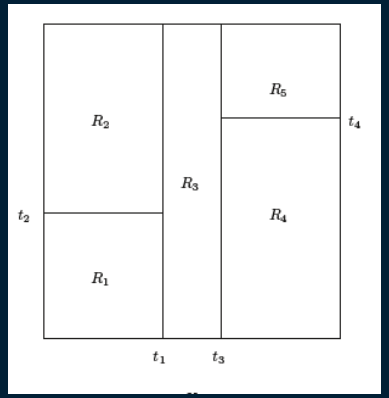
Interpretation

- For less experienced players, the number of hits in the previous year does not have much effect on their salary.
- For more experienced players, having more hits the previous year leads to higher salaries.
- While regression trees are not the most powerful predictive model in terms of accuracy, they are easy to interpret and have a nice graphical representation.

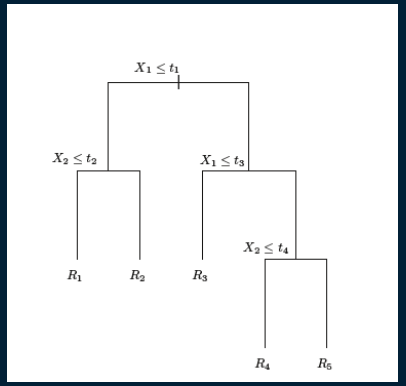


Making Predictions

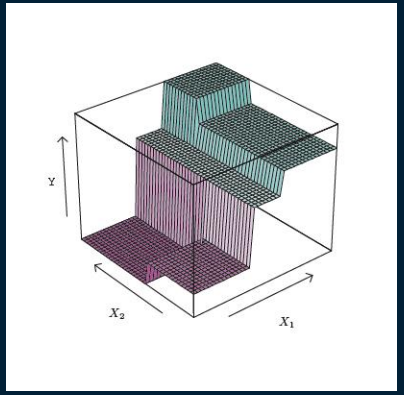
- For continuous targets, we predict the response for a given observation using the mean of the training observations in the region to which that test observation belongs.



Sample output of recursive binary splitting on a two-dimensional example.



Tree corresponding to this splitting



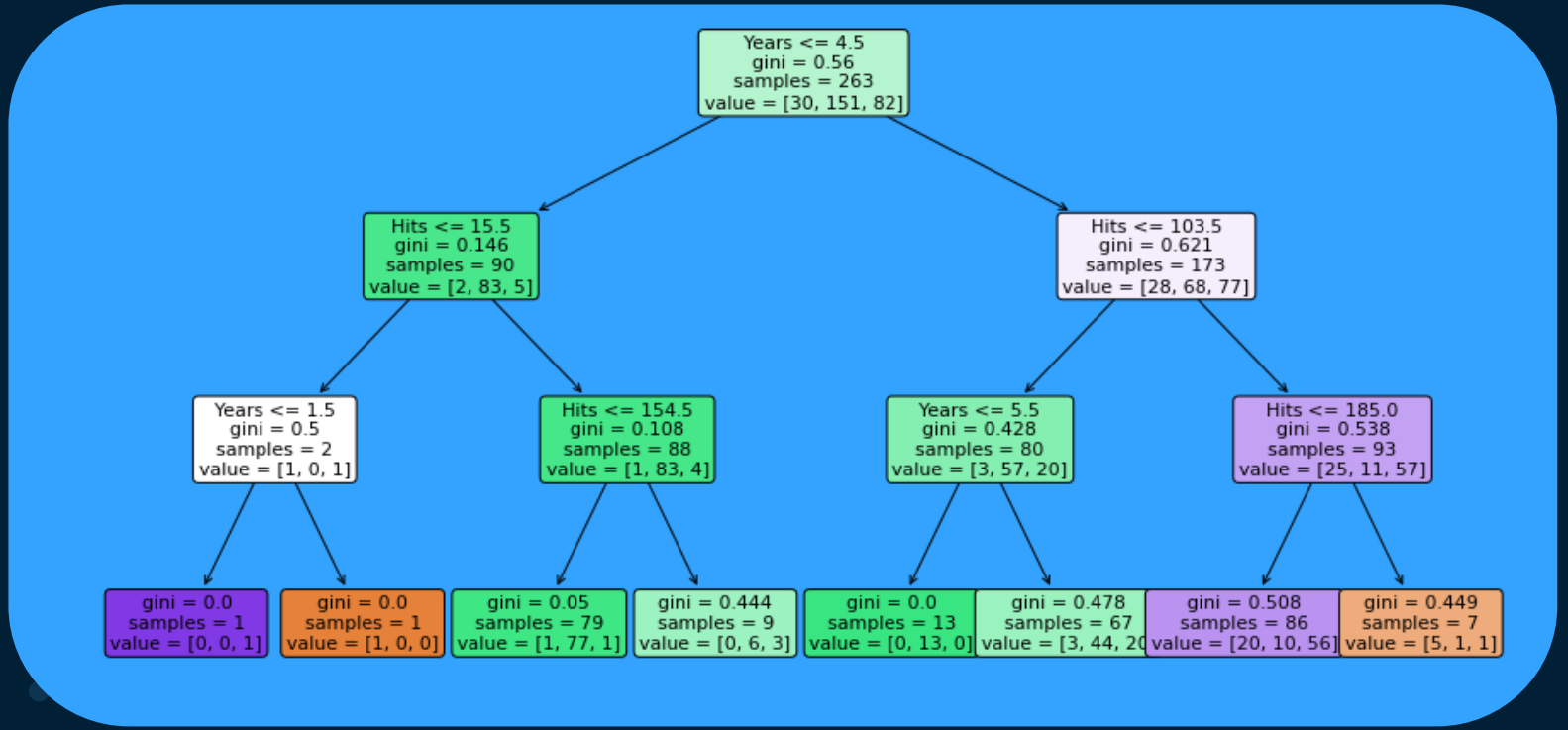
Perspective plot of prediction surface corresponding to this tree

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, or the mode, of training observations in the region to which it belongs.

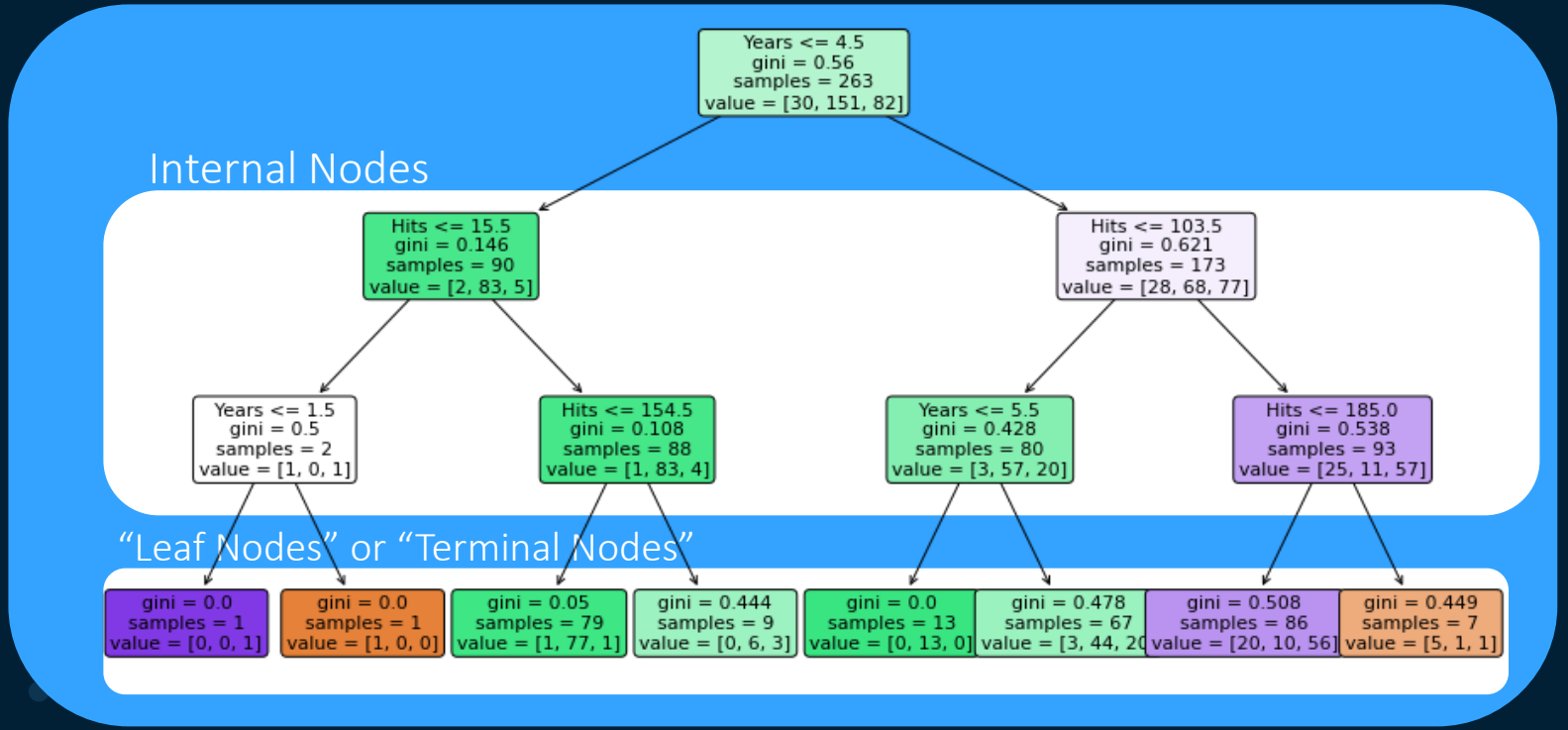
Classification Tree

Baseball Hitters Example



Terminology

Baseball Hitters Example



Constructing Trees

Constructing Trees

Objective

- The objective in constructing trees is to divide the "predictor space" – the n-dimensional space of all possible values of all n predictors – into non-overlapping regions that are as "pure" as possible with regards to the target attribute.
- Key questions:
  - How to define "purity"?
  - How and where to draw the boundaries?
  - How many regions should we split the predictor space into?

Constructing Trees

Variants of Trees

Details of algorithm depends on two factors

- Attribute types
  - Binary
  - Nominal
  - Ordinal
  - Continuous
- Split types
  - 2-way split
  - Multi-way split

Constructing Trees

Defining Purity

Numerical target attribute

- Residual sum of squared differences:

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

Average target attribute for elements in the region

Categorical target attribute

- Classification error rate:

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

Proportion of observations in region m from k^{th} class

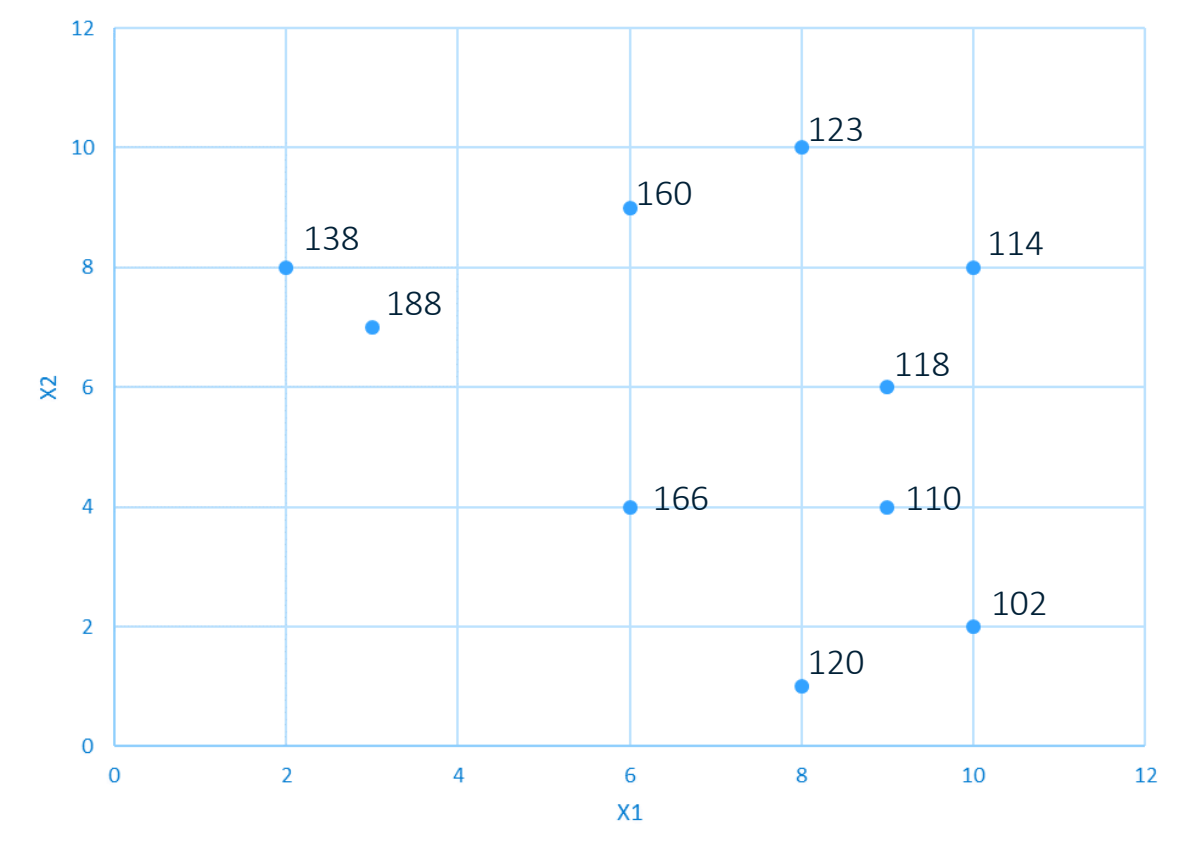
- Gini index:

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

- Entropy:

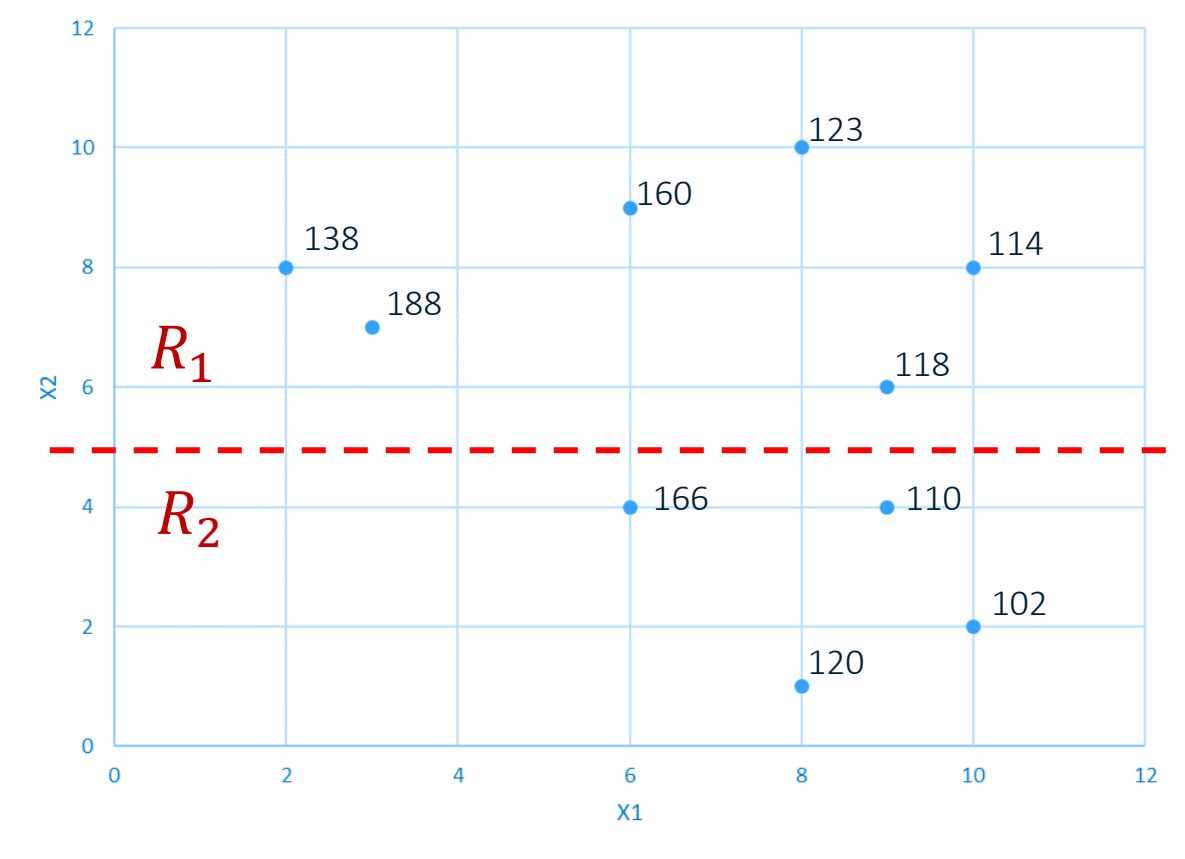
$$D = -\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}$$

Calculating "Purity"



From visual inspection, where would you draw a line (horizontally or vertically) to achieve maximum purity?

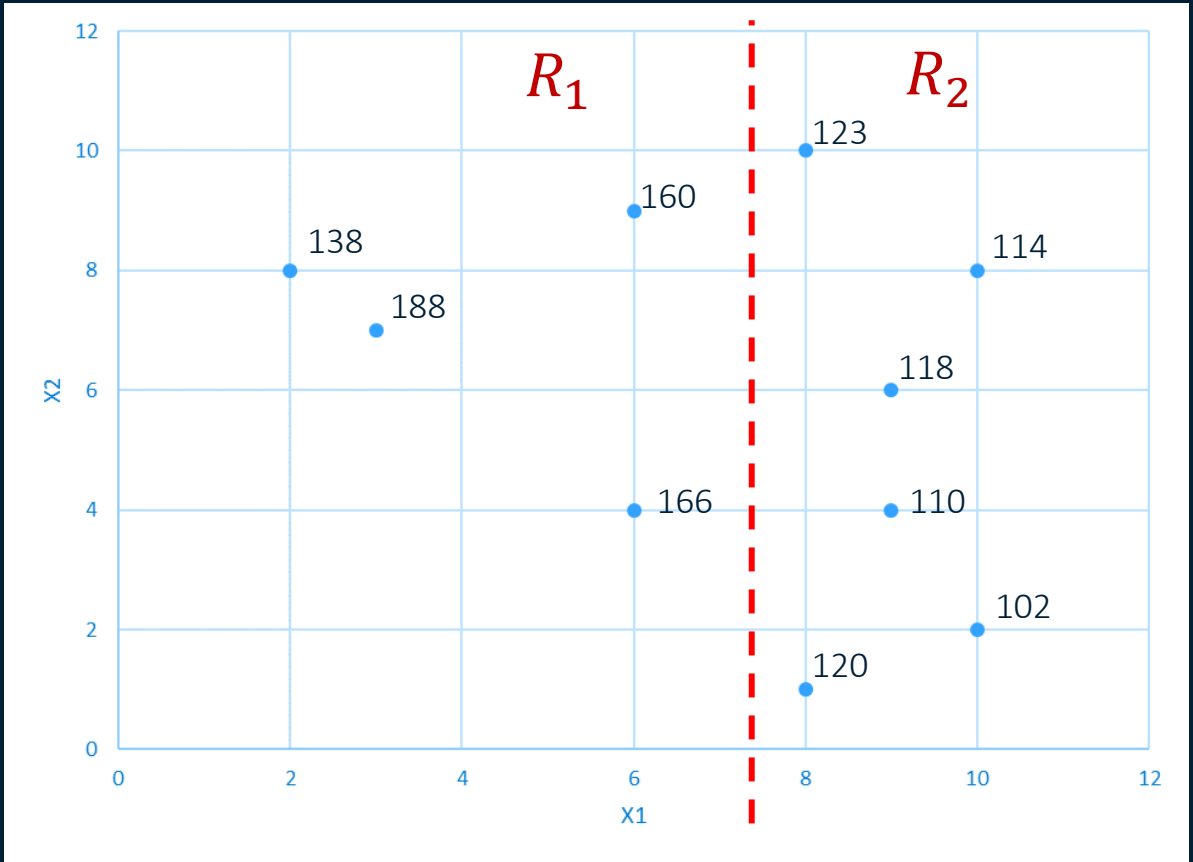
Calculating "Purity"



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

$$4156.8 + 614.8 = 4771.6$$

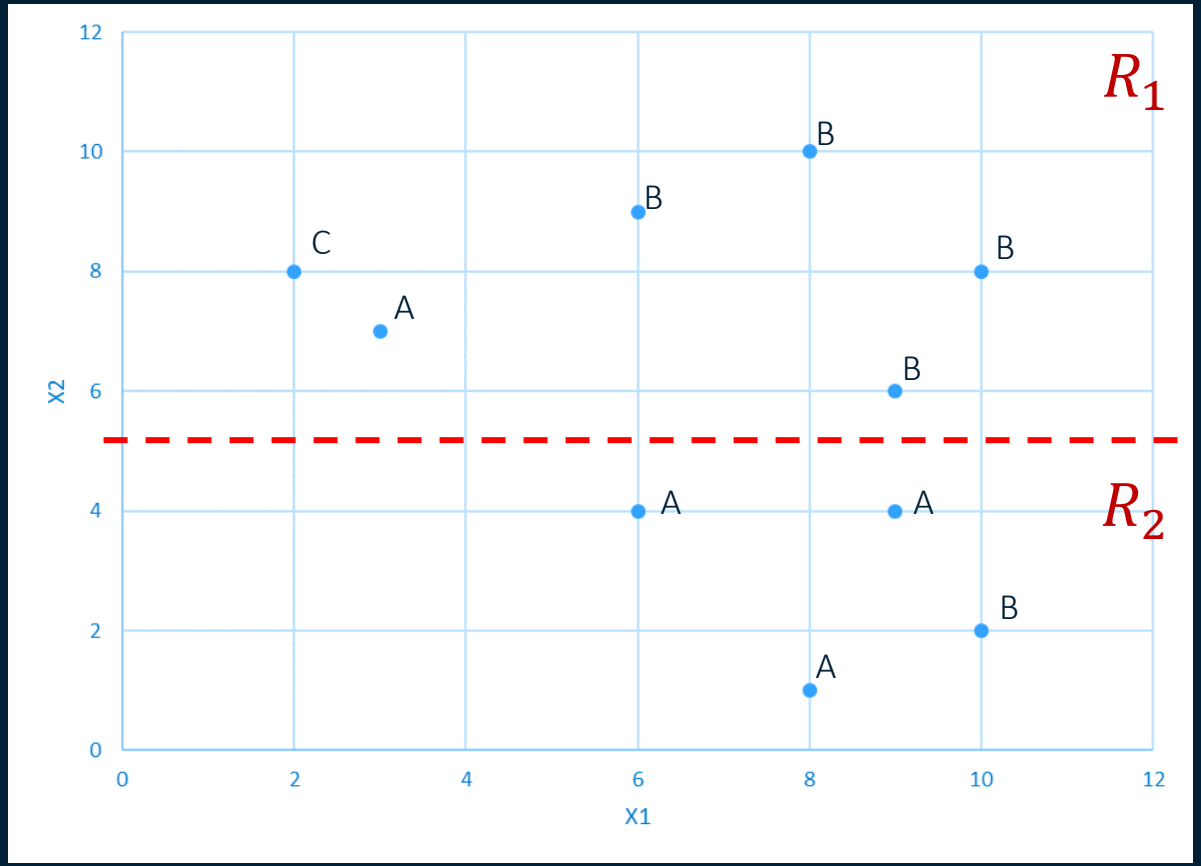
Calculating "Purity"



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

$$1268 + 291.5 = 1559.5$$

Calculating "Purity"



Classification Error Rate

$\text{Max}(\hat{p}_{1k}) = 1 - 0.66 = 0.33$

$\text{Max}(\hat{p}_{2k}) = 1 - 0.75 = 0.25$

Gini Index

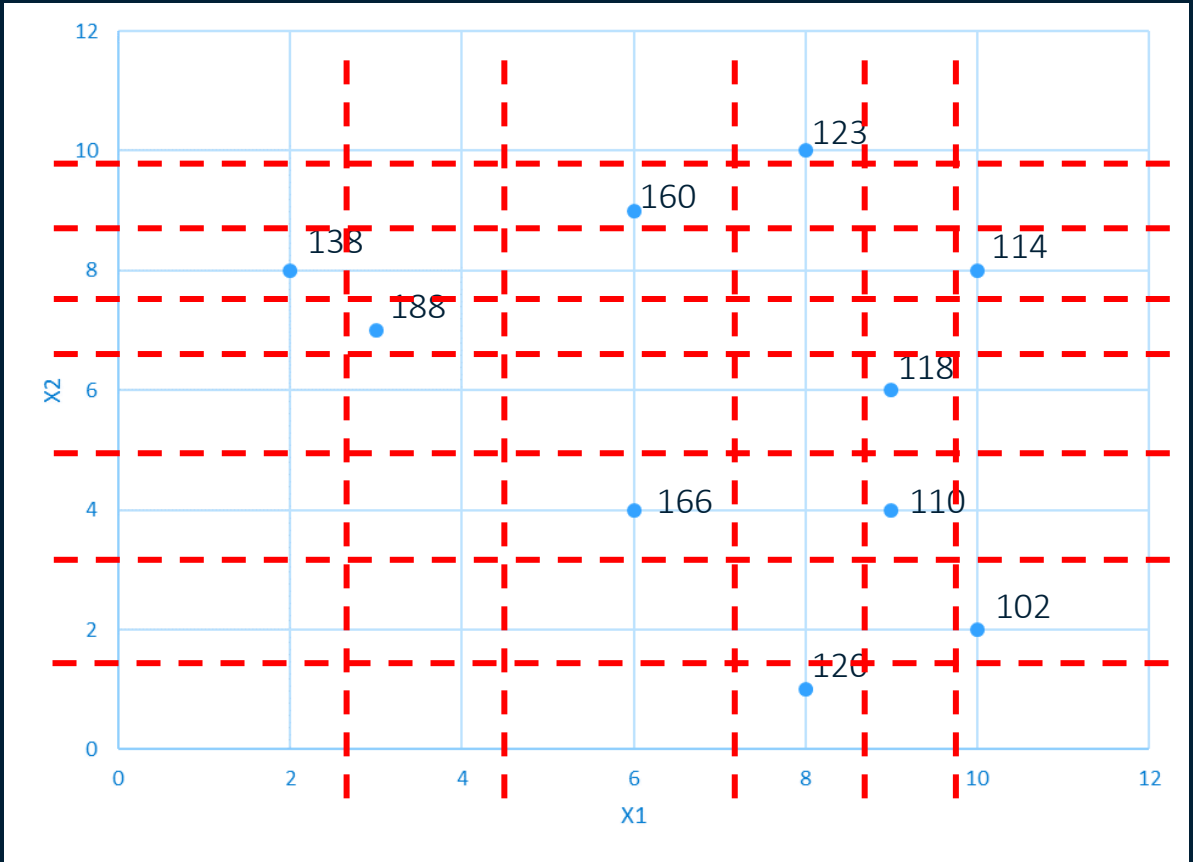
$$G_1 = \left(\frac{1}{6}\right)\left(\frac{5}{6}\right) + \left(\frac{1}{6}\right)\left(\frac{5}{6}\right) + \left(\frac{4}{6}\right)\left(\frac{2}{6}\right)$$

$$G_2 = \left(\frac{1}{4}\right)\left(\frac{3}{4}\right) + \left(\frac{3}{4}\right)\left(\frac{1}{4}\right)$$

Entropy

$$D_1 = -(2 * \left(\frac{1}{6}\right)\log\left(\frac{1}{6}\right) + \left(\frac{4}{6}\right)\log\left(\frac{4}{6}\right))$$

How and Where to Draw Boundaries



For simplicity and ease of interpretation and computation, we limit our selection of regions to boxes (not circles or other non-linear boundaries.

Where and How to Draw Boundaries

Tree Building Algorithm

- It is computationally infeasible to consider every possible partition of the feature space.
- Standard algorithm is known as recursive binary splitting
  - Begin at the top of the tree and successively split the predictor space into two new branches resulting in the highest overall purity of the nodes
  - Repeat using one of the regions based on overall purity until a stopping condition is reached (see next section)
- This algorithm is referred to as a "greedy" algorithm because it makes locally optimum choices instead of searching for a global optimum

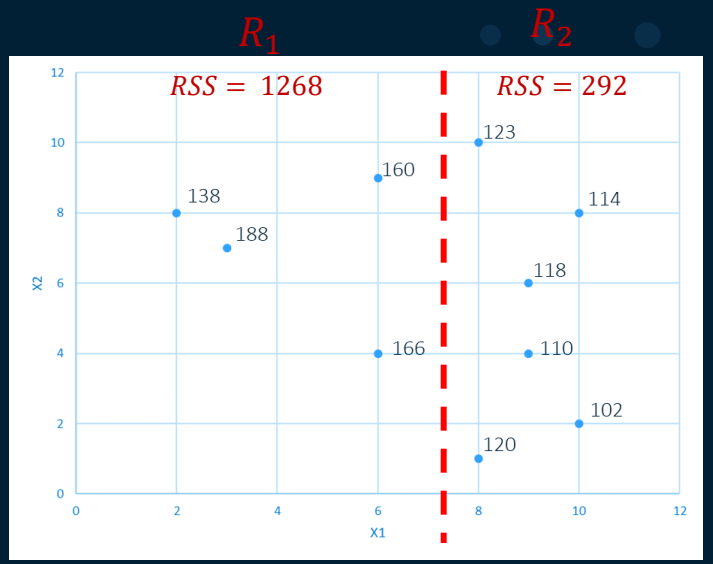
Example

Find First Cutpoint

Candidate Cutpoints

```
> x1_cutpoints
# A tibble: 6 x 6
     x1 Size_Left Size_Right RSS_Left RSS_Right TotRSS
  <dbl>     <int>      <int>    <dbl>     <dbl>  <dbl>
1     2         1          9        0      7186.  7186.
2     6         4          6     1268       292.  1560.
3     9         8          2     5456.        72  5528.
4     8         6          4     3569.       140  3709.
5    10        10          0     7205.         0  7205.
6     3         2          8     1250      3838.  5088.
```

```
> x2_cutpoints
# A tibble: 6 x 6
     x2 Size_Left Size_Right RSS_Left RSS_Right TotRSS
  <dbl>     <int>      <int>    <dbl>     <dbl>  <dbl>
1     8         8          2     6376       684.  7060.
2     9         9          1     7073.        0   7073.
3     6         5          5     2493.     3567.  6060
4    10        10          0     7205.         0  7205.
5     2         2          8      162      5732.  5894.
6     7         6          4     5992      1213.  7205.
```



- Total RSS before cut: 7205
- Total RSS after cut: 1268 + 292 = 1560

Example

Find Second Cutpoint

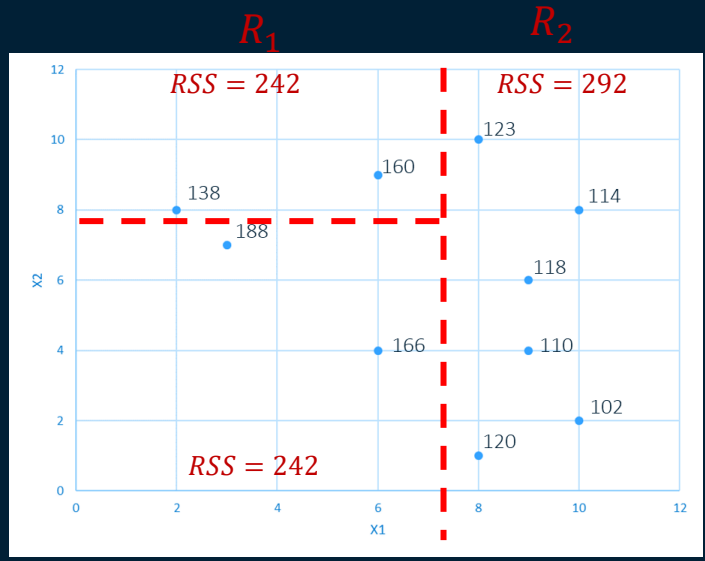
Candidate Cutpoints

```
> R1
# A tibble: 4 x 3
     x1    x2     y
  <dbl> <dbl> <dbl>
1     2     8   138
2     6     9   160
3     6     4   166
4     3     7   188
```

```
> R2
# A tibble: 6 x 3
     x1    x2     y
  <dbl> <dbl> <dbl>
1     9     6   118
2     8    10   123
3    10     2   102
4    10     8   114
5     9     4   110
6     8     1   120
```

```
> R1x1_cutpoints
# A tibble: 3 x 6
     x1 Size_Left Size_Right RSS_Left RSS_Right TotRSS
  <dbl>     <int>      <int>    <dbl>     <dbl>  <dbl>
1     2         1          3        0      435.   435.
2     6         4          0     1268        0   1268
3     3         2          2     1250       18   1268
> R1x2_cutpoints
# A tibble: 4 x 6
     x2 Size_Left Size_Right RSS_Left RSS_Right TotRSS
  <dbl>     <int>      <int>    <dbl>     <dbl>  <dbl>
1     8         3          1     1256        0   1256
2     9         4          0     1268        0   1268
3     4         1          3        0     1256   1256
4     7         2          2      242      242    484
```

```
> R2x1_cutpoints
# A tibble: 3 x 6
     x1 Size_Left Size_Right RSS_Left RSS_Right TotRSS
  <dbl>     <int>      <int>    <dbl>     <dbl>  <dbl>
1     9         4          2     92.8        72   165.
2     8         2          4      4.5       140   144.
3    10         6          0     292.         0   292.
> R2x2_cutpoints
# A tibble: 6 x 6
     x2 Size_Left Size_Right RSS_Left RSS_Right TotRSS
  <dbl>     <int>      <int>    <dbl>     <dbl>  <dbl>
1     6         4          2      203      40.5   244.
2    10         6          0     292.        0   292.
3     2         2          4      162      92.8   255.
4     8         5          1     205.        0   205.
5     4         3          3     163.      40.7   203.
6     1         1          5        0      255.   255.
```



Total RSS before cut: 1560

Total RSS after cut: 242 + 242 + 292 = 776

Note: if we had selected the R1X1 cutpoint of 2, the total RSS after cut would have been 435 + 242 + 292 = 969

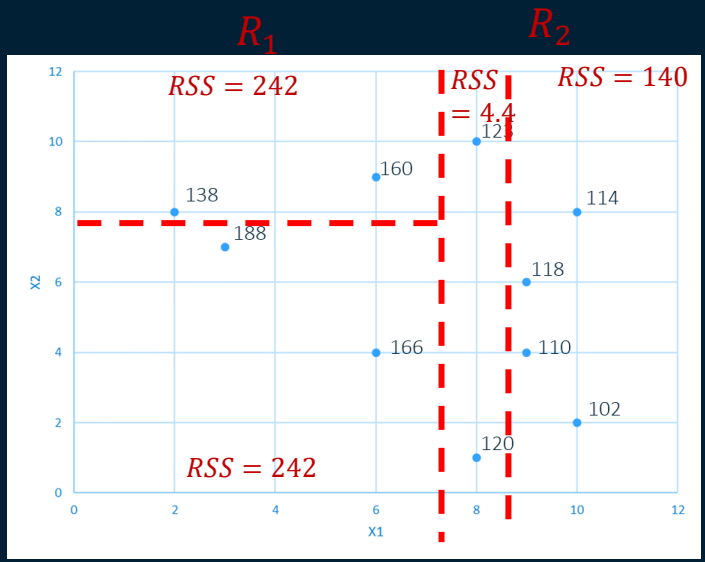
Example

Find Third Cutpoint

Candidate Cutpoints

```
> R2
# A tibble: 6 x 3
     x1    x2     y
  <dbl> <dbl> <dbl>
1     9     6   118
2     8    10   123
3    10     2   102
4    10     8   114
5     9     4   110
6     8     1   120
```

```
> R2x1_cutpoints
# A tibble: 3 x 6
     x1 Size_Left Size_Right RSS_Left RSS_Right TotRSS
  <dbl>     <int>      <int>    <dbl>     <dbl>  <dbl>
1     9         4          2     92.8        72   165.
2     8         2          4      4.5       140   144.
3    10         6          0    292.         0   292.
> R2x2_cutpoints
# A tibble: 6 x 6
     x2 Size_Left Size_Right RSS_Left RSS_Right TotRSS
  <dbl>     <int>      <int>    <dbl>     <dbl>  <dbl>
1     6         4          2      203      40.5   244.
2    10         6          0      292.      0     292.
3     2         2          4      162      92.8   255.
4     8         5          1      205.     0     205.
5     4         3          3      163.     40.7   203.
6     1         1          5        0     255.   255.
```



Total RSS before cut: 776

Total RSS after cut: 242 + 242 + 4.4 + 140 = 628

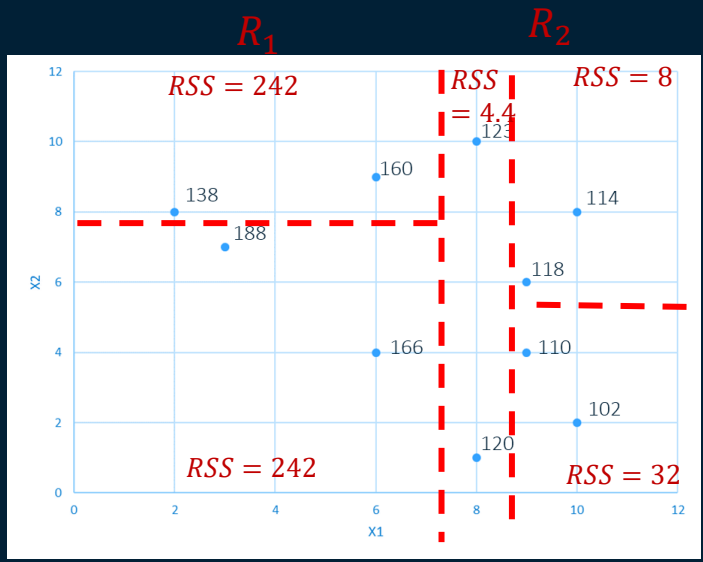
Example

Find Fourth Cutpoint

Candidate Cutpoints

```
> R2
# A tibble: 6 x 3
     x1    x2     y
  <dbl> <dbl> <dbl>
1     9     6   118
2     8    10   123
3    10     2   102
4    10     8   114
5     9     4   110
6     8     1   120
```

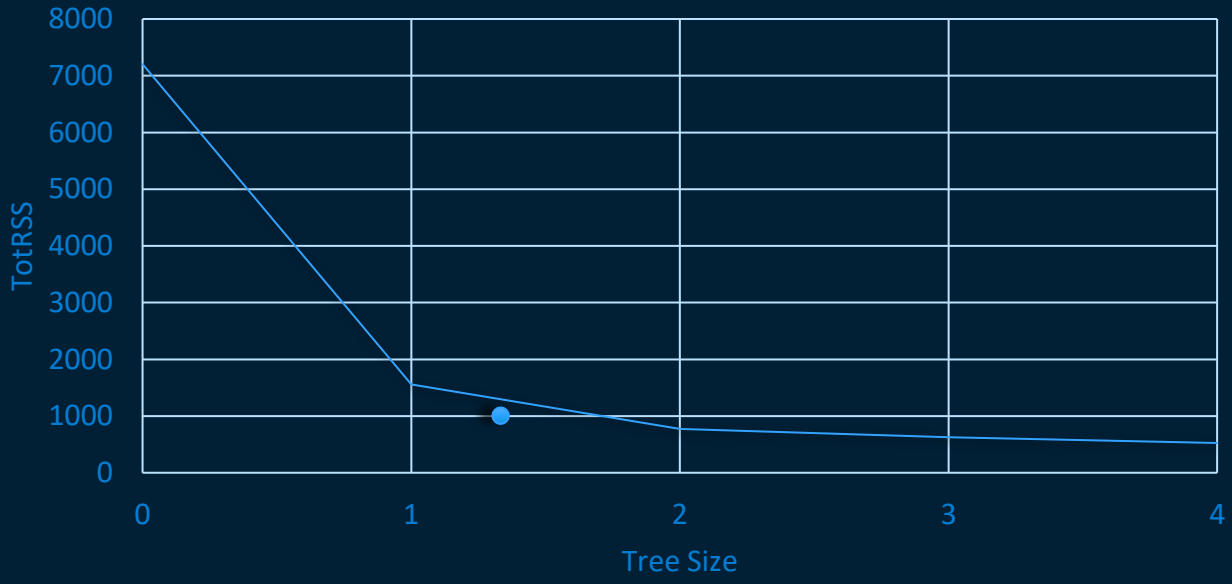
```
> R2x1_cutpoints
# A tibble: 3 x 6
     x1 Size_Left Size_Right RSS_Left RSS_Right TotRSS
  <dbl>     <int>      <int>    <dbl>     <dbl>  <dbl>
1     9         4          2     92.8        72   165.
2     8         2          4      4.5       140   144.
3    10         6          0    292.         0   292.
> R2x2_cutpoints
# A tibble: 6 x 6
     x2 Size_Left Size_Right RSS_Left RSS_Right TotRSS
  <dbl>     <int>      <int>    <dbl>     <dbl>  <dbl>
1     6         4          2      203      40.5   244.
2    10         6          0      292.      0     292.
3     2         2          4      162      92.8   255.
4     8         5          1      205.     0     205.
5     4         3          3      163.     40.7   203.
6     1         1          5        0     255.   255.
```



Total RSS before cut: 628

Total RSS after cut: 242 + 242 + 4.4 + 8 + 32 = 528

Tree Size vs Total RSS



Tree Size Decisions

It’s necessary to decide how deep to make the tree (when to “stop”) or we would end up with every observation in its own leaf node

- Stopping conditions are generally set based on a maximum tree depth and/or a minimum node size
- It is also frequently necessary to limit the tree size to avoid overfitting and to increase interpretability.
  - To understand this, we must first discuss variance and bias in predictive models

Pruning Trees

- Objective: A smaller tree with fewer splits might lead to lower variance and better interpretation at the cost of a little bias.
- One approach: stop growing the tree when the decrease in RSS is less than some threshold
  - May miss very good splits further down the tree
- Alternate Approach: Grow a very large tree and then prune it back in order to obtain a subtree.

Overall objective is to reduce the tree size without reducing predictive accuracy as measured by a cross-validation set.

Pruning Trees

Cost Complexity Pruning ("Weakest Link Pruning")

Basic approach: modify the RSS algorithm that we are attempting to minimize by adding a "penalty" factor α (where $|T|$ is the number of nodes in the tree:

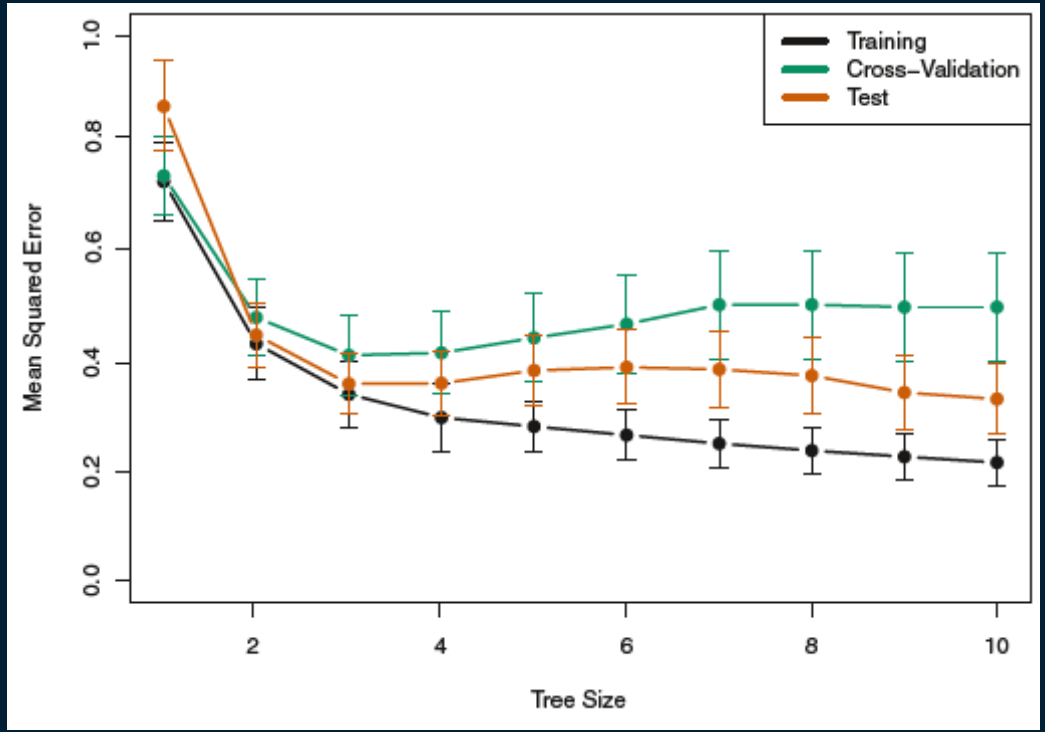
$$\sum_{m=1}^{|T|} \sum_{i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha |T|,$$

α is selected by a technique known as cross-validation

Tree-Pruning Algorithm

1. Grow a large tree stopping only when each terminal node has fewer than some minimum number of observations
2. Perform tuning to obtain a collection of best subtrees as a function of α
3. Perform K-fold cross-validation to choose α. For each $k = 1, \dots, K$:
   - Repeat seps 1 and 2 on the $\frac{k-1}{k}\text{th}$ fraction of the training data
   - Evaluate the RSS prediction error on the data in the left-out kth partition as a function of α
4. Return the subtree from Step 2 that corresponds to the chosen value of α

How Much to Prune?



Tree-Building Algorithms

- ID3 (1986): Creates multi-way trees using categorical variables
- C4.5: Successor to ID3 removes restriction that features must be categorical
- CART (Classification and Regression Trees). Similar to C4.5 but supports numerical response variables

Scikit-learn uses CART but does not currently support categorical variables (we must convert them to numerics)

Advantages and Disadvantages of Trees

- Easily explained
- Mirrors human decision-making processes
- Readily displayed graphically
- Handles qualitative predictors without requiring dummy variables
- Generally, not as accurate predictions as other techniques
- Have a tendency to overfit
- Data requires minimal pre-processing.
  - Automatically handles missing values, highly correlated predictors, and skewed variables

Scikit-Learn Decision Trees

Scikit-Learn Decision Trees

- Similar to the sklearn linear models, there are two standard classes:
  - DecisionTreeClassifier
  - DecisionTreeRegressor
- These classes have similar methods and attributes as the LinearRegression class:
  - .fit(x,y)
  - .predict(X)
  - .predict_proba(X)
  - .score(X,y)

Sklearn Decision Tree Classifier

Common parameters available when defining a decision tree classifier model:

- criterion (gini, entropy, etc.)
- max_depth: Maximum tree depth
- min_samples Minimum number of samples required to split a node
- max_leaf_node: Maximum number of leaf nodes
- ccp_alpha: Cost complexity parameter (alpha)

Fitting Classification Trees

Carseats Example

```
1 carseats = pd.read_csv('Carseats.csv').dropna()
2 carseats['high']= pd.Series(np.where(carseats.Sales<=8, 0, 1))  # Define category response - high sales
3 carseats
```

	Sales	CompPrice	Income	Advertising	Population	Price	ShelveLoc	Age	Education	Urban	US	high
0	9.50	138	73	11	276	120	Bad	42	17	Yes	Yes	1
1	11.22	111	48	16	260	83	Good	65	10	Yes	Yes	1
2	10.06	113	35	10	269	80	Medium	59	12	Yes	Yes	1
3	7.40	117	100	4	466	97	Medium	55	14	Yes	Yes	0
4	4.15	141	64	3	340	128	Bad	38	13	Yes	No	0
...	...	...	...	...	...	...	...	...	...	...	...	...
395	12.57	138	108	17	203	128	Good	33	14	Yes	Yes	1
396	6.14	139	23	3	37	120	Medium	55	11	No	Yes	0
397	7.41	162	26	12	368	159	Medium	40	18	Yes	Yes	0
398	5.94	100	79	7	284	95	Bad	50	12	Yes	Yes	0
399	9.71	134	37	0	27	120	Good	49	16	Yes	Yes	1

400 rows × 12 columns

Fitting Classification Trees

Converting Categories to Numeric Variables

```
carseats['US'] = pd.get_dummies(carseats['US'])['Yes']
carseats['Urban'] = pd.get_dummies(carseats['Urban'])['Yes']
carseats.loc[carseats['ShelveLoc'] == 'Bad','ShelveLoc'] = 0
carseats.loc[carseats['ShelveLoc'] == 'Medium', 'ShelveLoc'] = 1
carseats.loc[carseats['ShelveLoc'] == 'Good', 'ShelveLoc'] = 2
carseats
```

	Sales	CompPrice	Income	Advertising	Population	Price	ShelveLoc	Age	Education	Urban	US	high
0	9.50	138	73	11	276	120	0	42	17	1	1	1
1	11.22	111	48	16	260	83	2	65	10	1	1	1
2	10.06	113	35	10	269	80	1	59	12	1	1	1
3	7.40	117	100	4	466	97	1	55	14	1	1	0
4	4.15	141	64	3	340	128	0	38	13	1	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...
395	12.57	138	108	17	203	128	2	33	14	1	1	1
396	6.14	139	23	3	37	120	1	55	11	0	1	0
397	7.41	162	26	12	368	159	1	40	18	1	1	0
398	5.94	100	79	7	284	95	0	50	12	1	1	0
399	9.71	134	37	0	27	120	2	49	16	1	1	1

400 rows × 12 columns

Fitting Classification Trees

No Limits on Depth and Alpha = 0 (no cost complexity pruning)

```
X= carseats.drop(['high','Sales'],axis=1)
y= carseats['high']
X_train, X_test, y_train, y_test= train_test_split(X,y, test_size=0.5, random_state=0)
```

Built classification tree with no limit on tree depth

```
carseats_classifier= DecisionTreeClassifier()
carseats_classifier.fit(X_train,y_train)
print(tree.export_text(carseats_classifier, feature_names = list(X.columns)))
```

```
|--- Price <= 94.50
|   |--- Age <= 55.50
|   |   |--- ShelveLoc <= 0.50
|   |   |   |--- Advertising <= 2.50
|   |   |   |   |--- CompPrice <= 126.00
|   |   |   |   |   |--- class: 0
|   |   |   |   |--- CompPrice >  126.00
|   |   |   |   |   |--- class: 1
|   |   |   |--- Advertising >  2.50
|   |   |   |   |--- class: 1
|   |   |--- ShelveLoc >  0.50
|   |   |   |--- class: 1
|   |--- Age >  55.50
|   |   |--- Price <= 80.00
|   |   |   |--- class: 1
|   |   |--- Price >  80.00
|   |   |   |--- CompPrice <= 112.00
|   |   |   |   |--- class: 0
|   |   |   |--- CompPrice >  112.00
|   |   |   |   |--- Education <= 17.50
|   |   |   |   |   |--- class: 1
|   |   |   |   |--- Education >  17.50
```

Fitting Classification Trees

Check Confusion Matrix and Misclassification Rates

```
#  Check training partition
pred= carseats_classifier.predict(X_train)
cm = pd.DataFrame(confusion_matrix(y_train, pred).T, index=['No', 'Yes'], columns=['No', 'Yes'])
cm.index.name = 'Predicted'
cm.columns.name = 'True'
cm
```

Predicted	True No	True Yes
No	118	0
Yes	0	82

```
#  Check test partition
pred= carseats_classifier.predict(X_test)
cm = pd.DataFrame(confusion_matrix(y_test, pred).T, index=['No', 'Yes'], columns=['No', 'Yes'])
cm.index.name = 'Predicted'
cm.columns.name = 'True'
cm
```

Predicted	True No	True Yes
No	94	34
Yes	24	48

```
misclass_rate = (cm["No"]['Yes'] + cm['Yes']['No'])/sum(cm['No'] + cm['Yes'])
print('Misclassification rate:', misclass_rate)
```

```
Misclassification rate: 0.29
```

Fitting Classification Trees

Perform Cost Complexity Pruning

- To help specify an optimal alpha for cost complexity pruning, sklearn provides a method “cost_complexity_pruning_path” that returns the “effective alpha” and the corresponding total leaf impurities at each step of the pruning process
  - Effective alpha – the alpha that would generate the next change in the split decisions

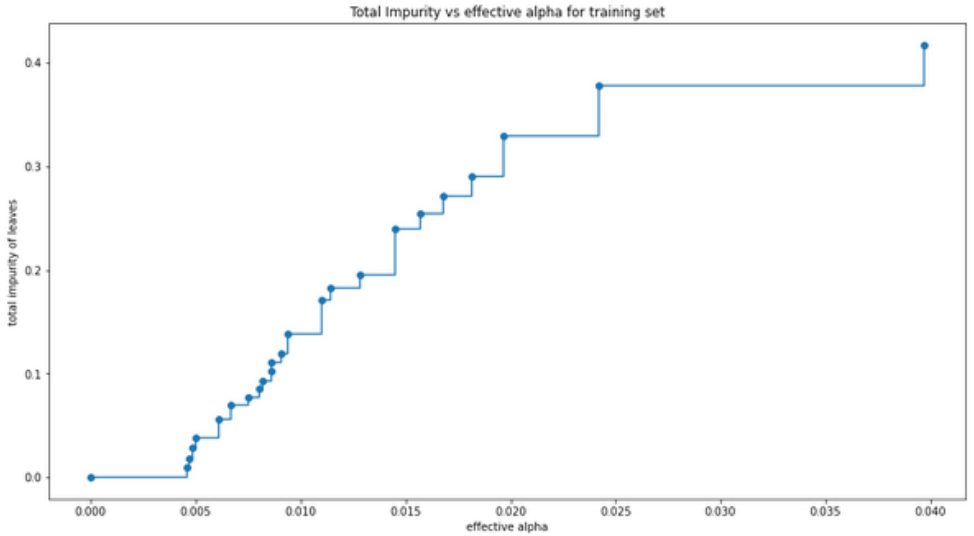
Perform Cost Complexity Pruning

Calculate and Plot Alphas and Corresponding Impurities

```
carseats_classifier_cv = DecisionTreeClassifier(random_state=0)
path = carseats_classifier_cv.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas, impurities = path.ccp_alphas, path.impurities
```

```
fig, ax = plt.subplots()
# Remove the last (highesst impurity) effective alpha because it corresponds to the case of a single node
ax.plot(ccp_alphas[:-1], impurities[:-1], marker="o", drawstyle="steps-post")
ax.set_xlabel("effective alpha")
ax.set_ylabel("total impurity of leaves")
ax.set_title("Total Impurity vs effective alpha for training set")
```

```
Text(0.5, 1.0, 'Total Impurity vs effective alpha for training set')
```



Perform Cost Complexity Pruning

Build a Classification Tree For Every Alpha

```
clfs = []
for ccp_alpha in ccp_alphas:
    clf = DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
    clf.fit(X_train, y_train)
    clfs.append(clf)
clfs
```

```
[DecisionTreeClassifier(random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.004583333333333333, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.004666666666666668, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.004838709677419352, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.004991134751773053, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.006086956521739295, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.006666666666666666, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.0075, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.007999999999999998, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.008166666666666666, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.008571428571428574, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.008571428571428574, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.009072580645161296, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.009360717100078804, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.011014492753623187, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.011428571428571427, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.012803030303030299, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.014496376811594202, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.015706521739130044, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.016805555555555546, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.018137035378414698, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.019649390243902436, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.024206720455228287, random_state=0),
 DecisionTreeClassifier(ccp_alpha=0.03970452508960581, random_state=0)]
```

Perform Cost Complexity Pruning

Remove the Last Alpha Option Because it is the Single Node Tree

Remove the last node becuase it is the trivial single-node case

```
print(
    "Number of nodes in the last tree is: {} with ccp_alpha: {}".format(
        clfs[-1].tree_.node_count, ccp_alphas[-1]
    )
)
```

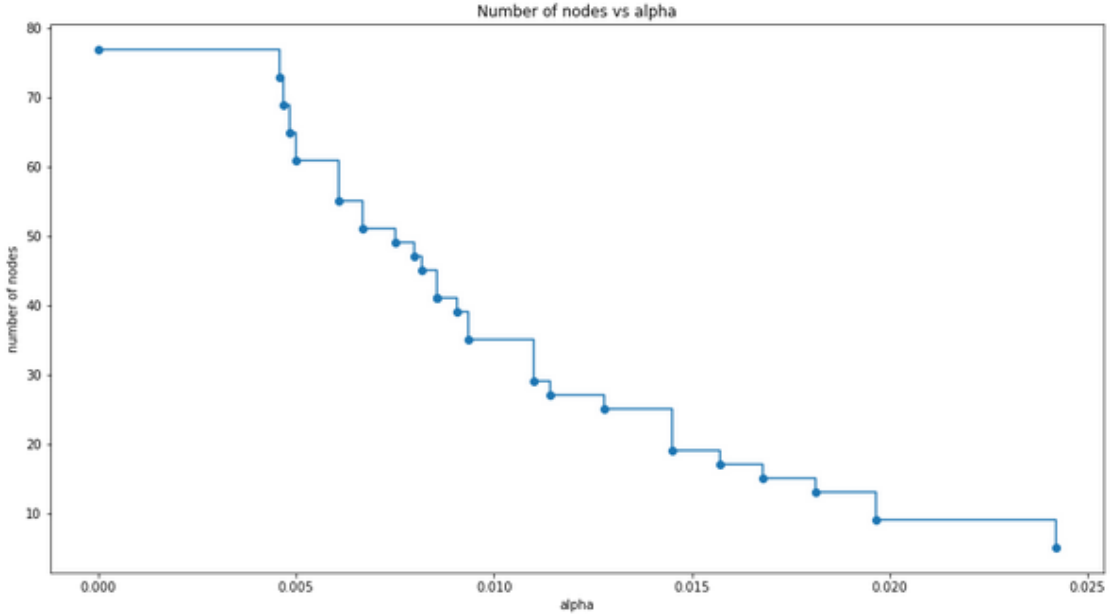
```
Number of nodes in the last tree is: 3 with ccp_alpha: 0.03970452508960581
```

```
clfs = clfs[:-1]
ccp_alphas = ccp_alphas[:-1]
```

Perform Cost Complexity Pruning

Plot Number of Nodes vs Alpha

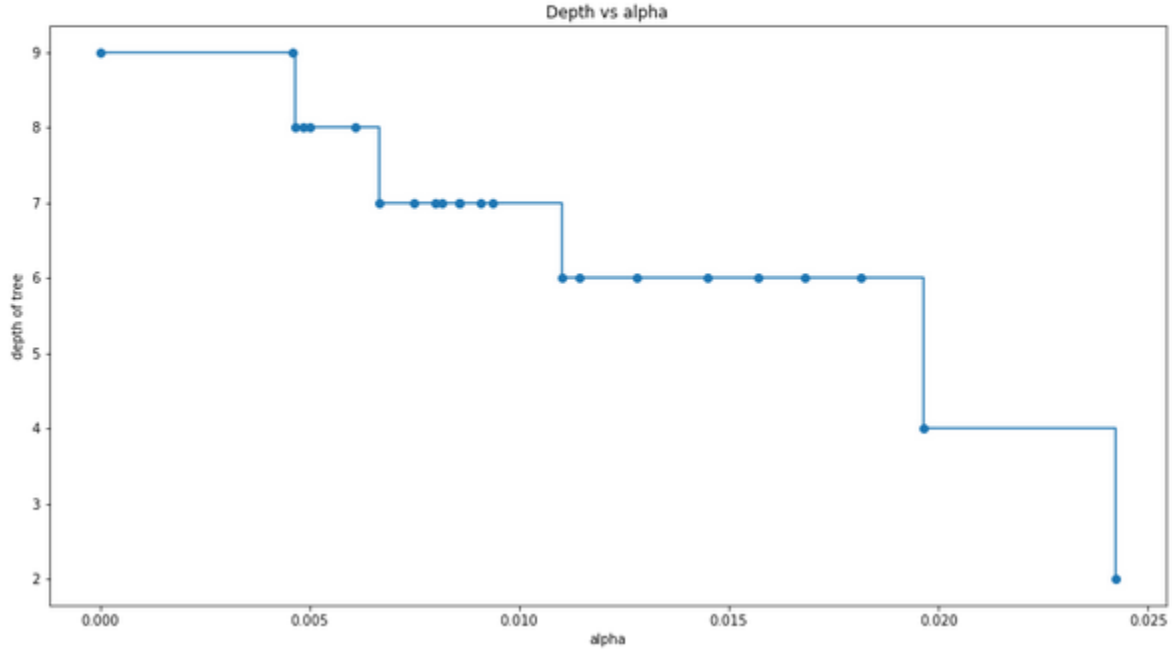
```
node_counts = [clf.tree_.node_count for clf in clfs]
depth = [clf.tree_.max_depth for clf in clfs]
plt.plot(ccp_alphas, node_counts, marker="o", drawstyle="steps-post")
plt.xlabel("alpha")
plt.ylabel("number of nodes")
plt.title("Number of nodes vs alpha")
plt.show()
```



Fitting Classification Trees

Plot Number of Levels vs Alpha

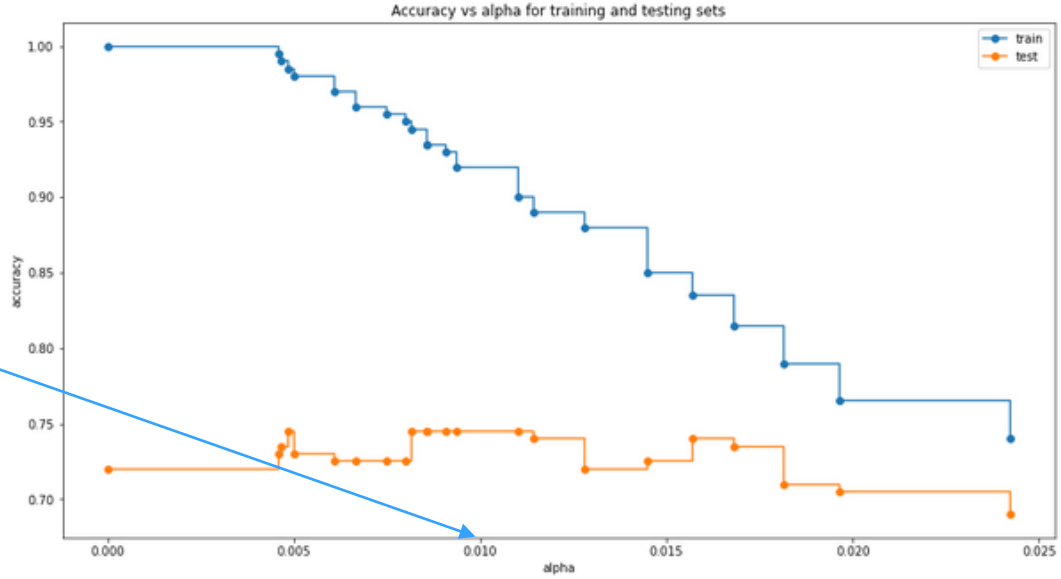
```
1 plt.plot(ccp_alphas, depth, marker="o", drawstyle="steps-post")
2 plt.xlabel("alpha")
3 plt.ylabel("depth of tree")
4 plt.title("Depth vs alpha")
5 plt.show()
```



Fitting Classification Trees

Plot accuracy vs alpha for training and test partitions

```
train_scores = [clf.score(X_train, y_train) for clf in clfs]
test_scores = [clf.score(X_test, y_test) for clf in clfs]
fig, ax = plt.subplots()
ax.set_xlabel("alpha")
ax.set_ylabel("accuracy")
ax.set_title("Accuracy vs alpha for training and testing sets")
ax.plot(ccp_alphas, train_scores, marker="o", label="train", drawstyle="steps-post")
ax.plot(ccp_alphas, test_scores, marker="o", label="test", drawstyle="steps-post")
ax.legend()
plt.show()
```



Let's try alpha = 0.01

Fitting Classification Trees

Try Re-Running With Alpha = 0.01

```
carseats_classifier= DecisionTreeClassifier(ccp_alpha = 0.01)
carseats_classifier.fit(X_train,y_train)
pred= carseats_classifier.predict(X_test)
cm = pd.DataFrame(confusion_matrix(y_test, pred).T, index=['No', 'Yes'], columns=['No', 'Yes'])
cm.index.name = 'Predicted'
cm.columns.name = 'True'
cm
```

Predicted \ True	No	Yes
No	98	31
Yes	20	51

```
misclass_rate = (cm["No"]['Yes'] + cm['Yes']['No'])/sum(cm['No'] + cm['Yes'])
print('Misclassification rate:', misclass_rate)
```

```
Misclassification rate: 0.255
```

Regression Trees

Boston Dataset

```
1 boston= pd.read_csv('Boston.csv', index_col=0)
2 boston.reset_index(drop=True, inplace=True)
3 boston
```

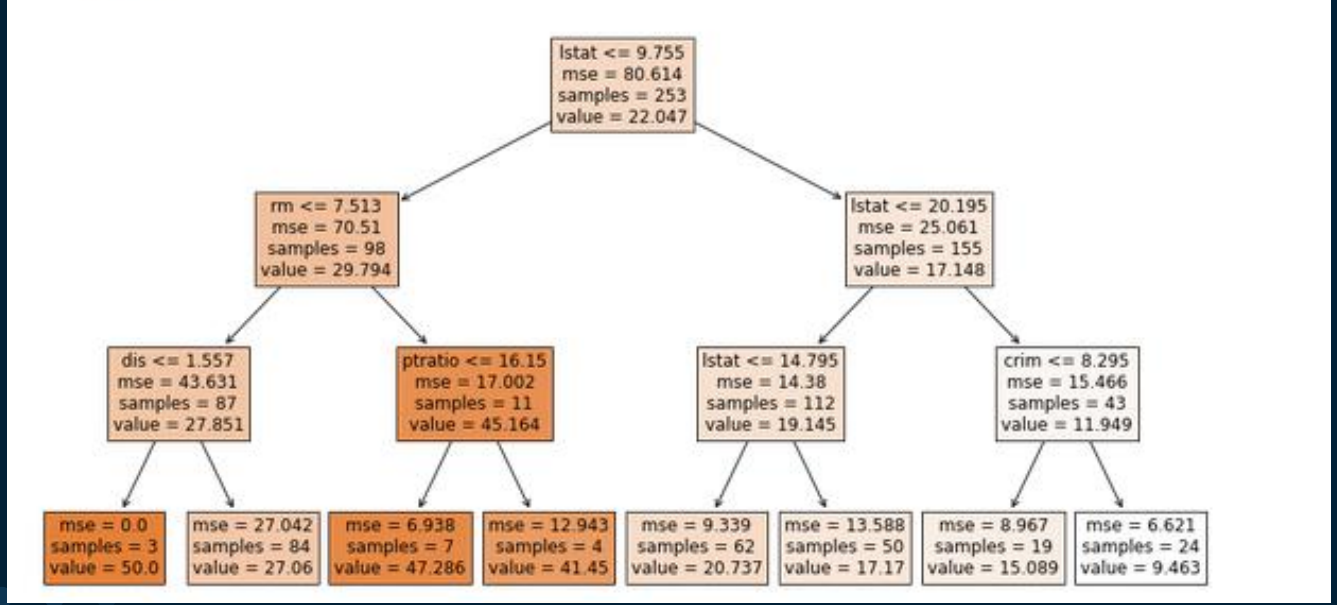
	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	lstat	medv
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	5.33	36.2
...	...	...	...	...	...	...	...	...	...	...	...	...	...
501	0.06263	0.0	11.93	0	0.573	6.593	69.1	2.4786	1	273	21.0	9.67	22.4
502	0.04527	0.0	11.93	0	0.573	6.120	76.7	2.2875	1	273	21.0	9.08	20.6
503	0.06076	0.0	11.93	0	0.573	6.976	91.0	2.1675	1	273	21.0	5.64	23.9
504	0.10959	0.0	11.93	0	0.573	6.794	89.3	2.3889	1	273	21.0	6.48	22.0
505	0.04741	0.0	11.93	0	0.573	6.030	80.8	2.5050	1	273	21.0	7.88	11.9

506 rows × 13 columns

Regression Trees

Fit Regression Tree with max_depth = 3

```
y= boston['medv']
x=  boston.drop('medv', axis=1)
x_train, x_test, y_train, y_test= train_test_split(x,y, test_size= 0.5, random_state =1)
boston_regressor = DecisionTreeRegressor(max_depth = 3)
boston_model= boston_regressor.fit(x_train, y_train)
tree.plot_tree(boston_regressor, feature_names = x_train.columns, fontsize = 12, filled = True)
plt.show()
```



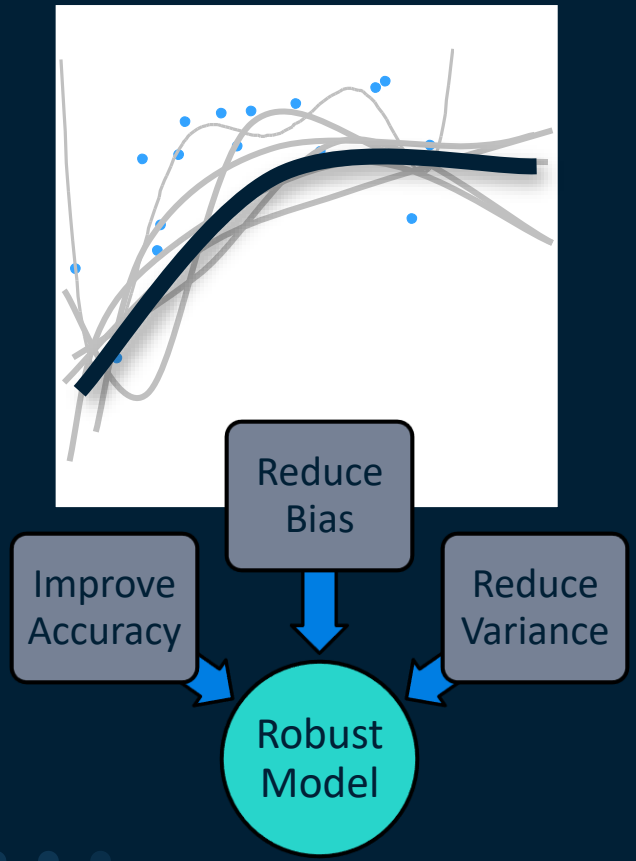
Decision Trees

Guidelines for Fitting

- Decision trees tend to overfit on data with a large number of features
- Performing dimensionality reduction (e.g., PCA) tends to get better results
- Understanding the decision tree structure is very valuable in performing inference
  - Start by limiting the depth (typically, max_depth = 3) to get a feel for the resulting logic
  - Gradually increase the depth while watching for overfitting
- If your dataset is significantly unbalanced (much more observations with one class), it can be balanced by stratified sampling

Ensemble Models

Ensemble Models



- An *ensemble model* is an aggregation of more than one model where the final prediction of the model is a combination of the predictions from the component models
- An ensemble model creates a single consensus prediction

Ensemble Models

Basic Types

- Averaging methods
  - Build several estimators independently and average their predictions
  - Objective is to decrease model variance
  - Examples include bagging and forests of randomized trees
- Boosting methods
  - Build estimators sequentially where the residuals from one model are used as inputs to the next model
  - Objective is to decrease bias of the combined estimator by combining several weak models to produce a powerful ensemble model
  - Examples include AdaBoost and Gradient Tree Boosting

Random Forests

Forest

- A *forest model* is an ensemble of classification or regression trees.
- The forest models were developed to overcome the instability that a single classification or regression tree exhibits with minor perturbations of the training data
- Trees in the forest differ from each other in two ways:
  - Training data for a tree is a sample with replacement from all observations.
  - Input variables considered for splitting a node are randomly selected from available inputs at each point in the algorithm

Decision Trees

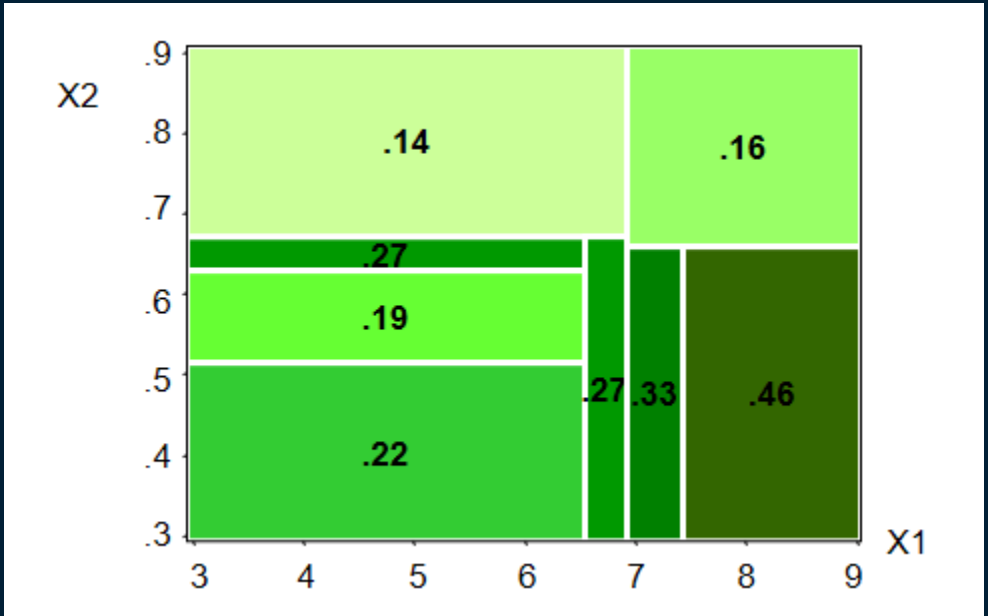
Reminder

- Repetitive partitioning of the predictor space into internal and leaf nodes.
- Leaf nodes yield Boolean prediction rules:

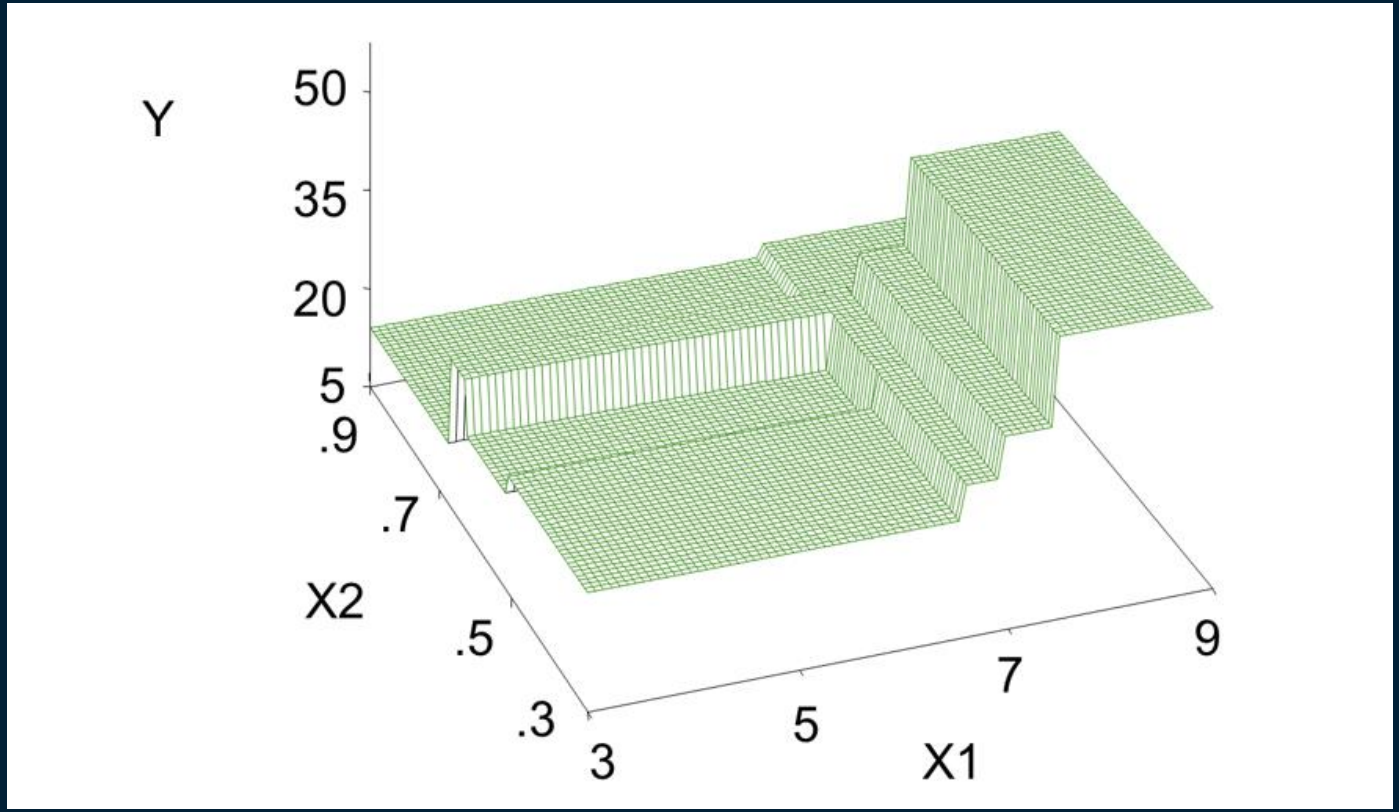
Leaf	X1	X2	Predicted Y
1	<6.5	<.51	.22
2	<6.5	[.51, .63)	.19
3	<6.5	[.63, .67)	.27
4	[6.5, 6.9)	<.67	.27
5	<6.9	≥.67	.14
6	[6.9, 7.4)	<.66	.33
7	≥7.4	<.66	.46
8	≥6.9	≥.66	.16

Decision Trees Reminder

Partitioned Input Space

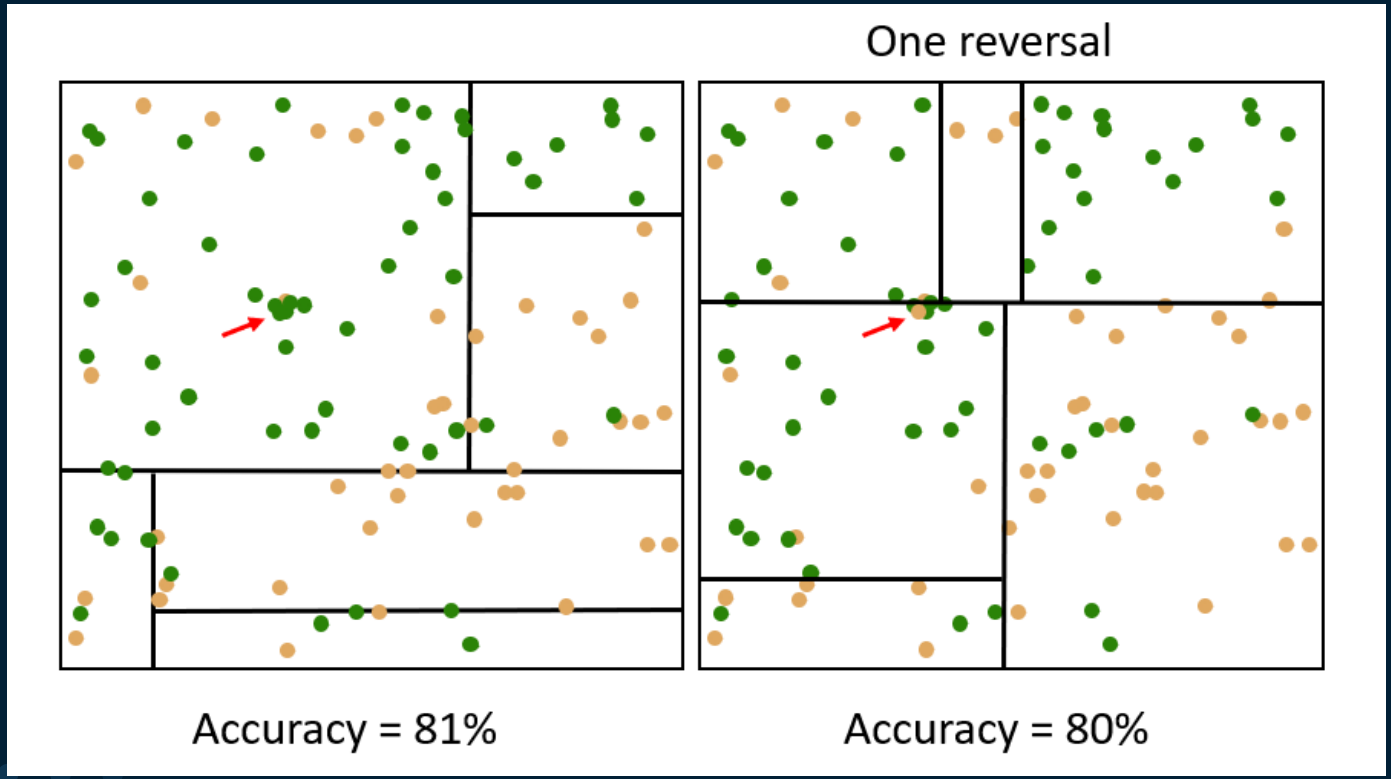


Multivariate Step Function

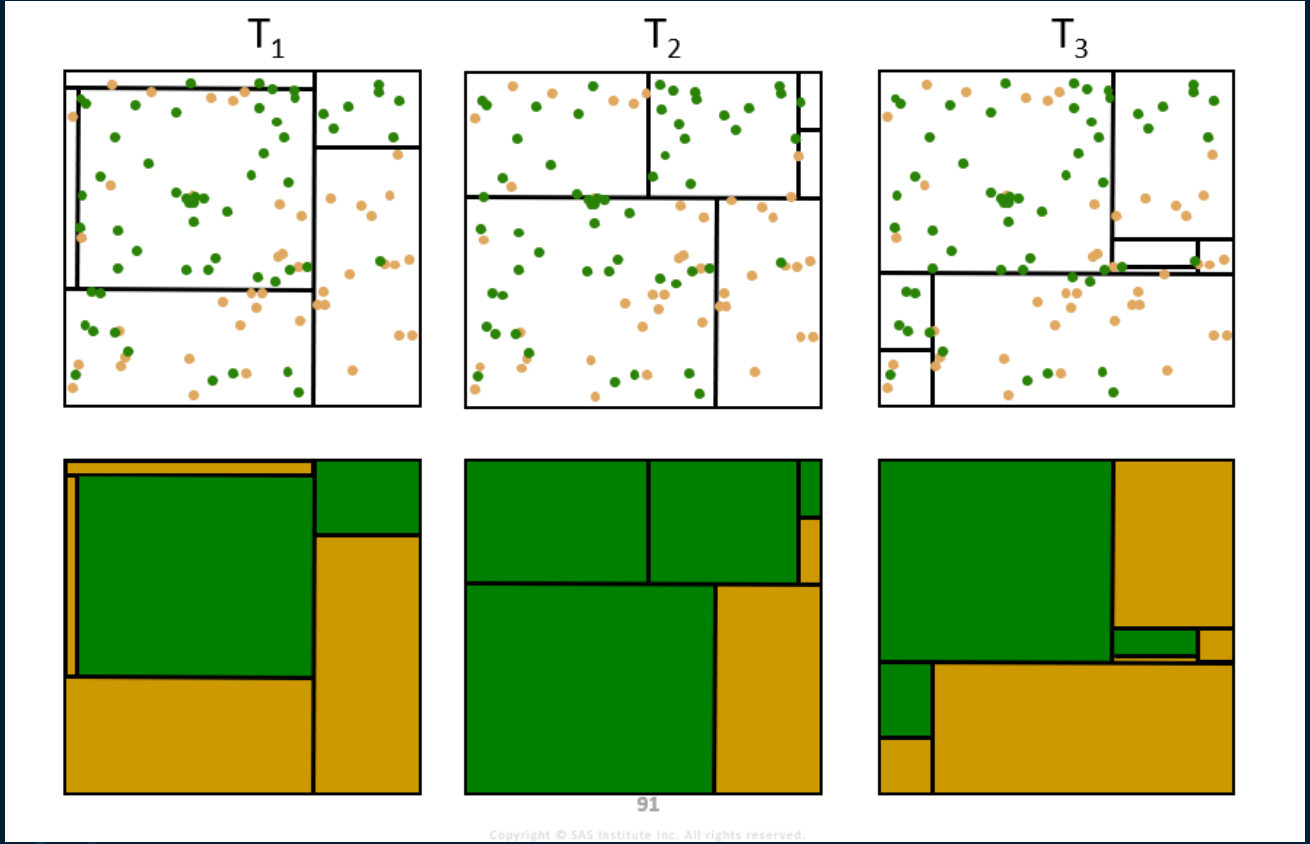


Classification Trees

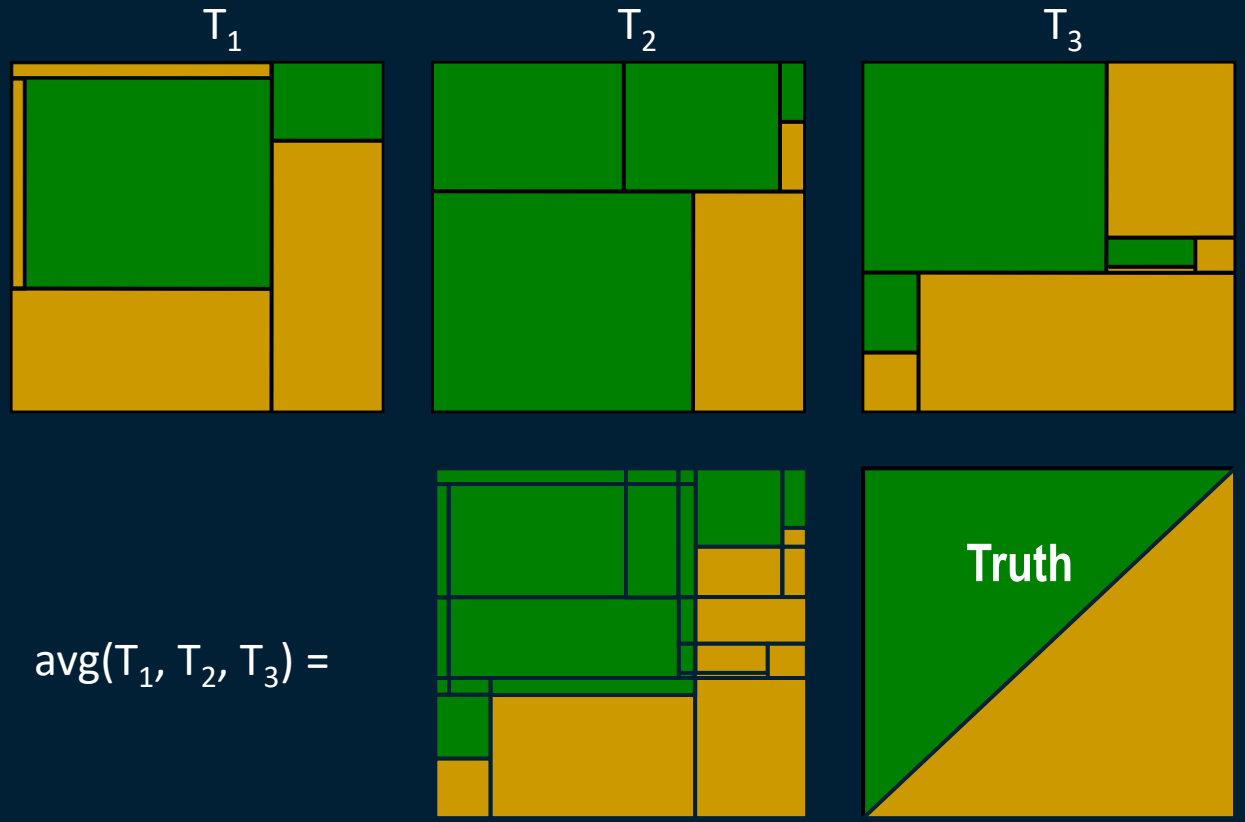
Instability



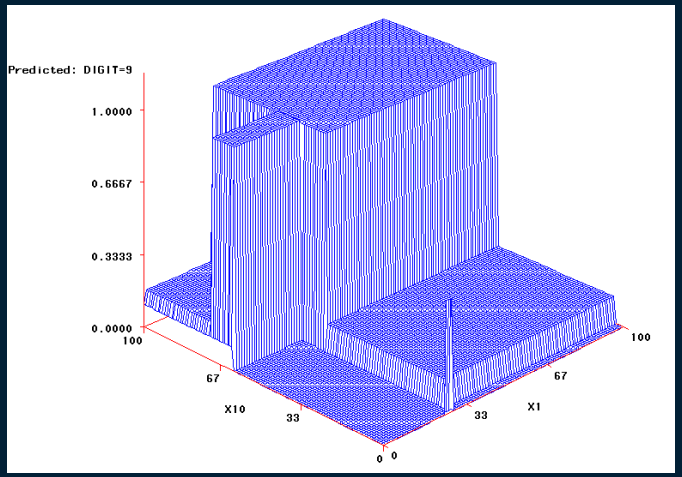
Perturb

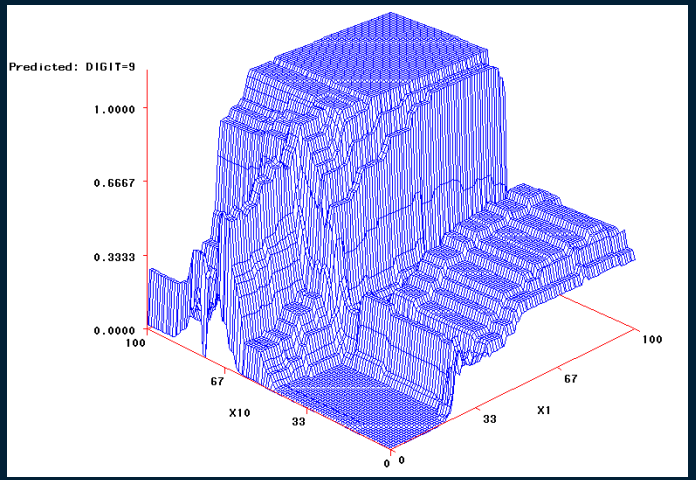


Combine



Single versus Bagged Trees





Ensemble Models - Review

Bootstrap Aggregation (Bagging)

- A general-purpose technique for reducing the variance of a predictive model.
  - Used with many predictive modeling types to reduce variance
- Recall from introductory statistics that if you take a set of n independent observations, each with variance σ^2, the variance of the mean of the observations is given by σ^2/n
- Bagging operates on the same principal by taking repeated samples with replacement from the training set

Ensemble Models - Review

Bagging Algorithm

- Draw a random sample of size N with replacements from the data consisting of N observations, referred to as a ***bootstrap sample***
- Construct a classification tree from the bootstrap sample
- Assign a class to each terminal node and store the predicted class of each observation
- Repeat steps 1-3 a large number of times
- Assign each observation to a final class by a majority vote over the set of trees

Ensemble Models - Review

Bootstrap Aggregation (Bagging)

- *Homogenous ensemble model*: the base learning algorithm is the same for all models (classification tree)
- Basic result: if N observations are drawn with replacement from N units, then 37% of the observations are left out on average

Forest Algorithm

- Forest algorithms extend the bagging technique to perform sampling of the rows ***and*** sampling of the columns at each step.
  - 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.
  - Typically, we choose $m \approx \sqrt{p}$
- The forest algorithm perturbs the training data more than the bagging algorithm, producing more variation among the trees in the ensemble.
- Ensembles of a more diverse set of trees often lead to improved predictive accuracy.

Rationale Behind Forests

- Suppose that we have a very strong predictor in the data set along with a number of other moderately strong predictors, then in the collection of bagged trees, most or all of them will use the very strong predictor for the first split
- All bagged trees will look similar. Hence all the predictions from the bagged trees will be highly correlated
- Averaging many highly correlated quantities does not lead to a large variance reduction, and thus random forests “de-correlates” the bagged trees leading to more reduction in variance

Forest Algorithm

- These are the three main options.
  - Number of trees
    - Specifies the number of trees that make up the forest
  - Number of predictors to split nodes
    - Specifies the number of input variables to consider splitting for each decision tree (Default = $\sqrt{\#\ of\ inputs}$)
  - Sampling strategy (Bootstrap)
    - Specifies the proportion of data to randomly sample for each tree (Default = 0.6)

Variable Importance

- Forests improve the performance of decision trees, but at a cost of easy visualization and interpretation
- Tree-based algorithms are able to calculate a variable importance measure by recording the total amount that the RSS is decreased due to splits over a given predictor (averaged over all B trees)

Summary

- Forests tend to give better prediction than any specific tree and often outperform other classes of models.
- Forests are challenging to interpret, but they can be considered an “ideal” model for other models to be compared against.
- Trees automatically handle missing values and variable reduction. Therefore, input data requires less preparation.
- Trees are independent of each other so they can be built simultaneously, making training faster.
- Forests are challenging to interpret
- Forest algorithm requires a large number of trees, which might make the algorithm slow for real-time prediction

Scikit-Learn Forests

Scikit-Learn Bagging Methods

Bagging Meta-Estimator

Bagging methods come in several variations:

- Random subsets of observations: Pasting
- Observations drawn with replacement: Bagging
- Random subsets of features: Random Subspaces
- Both random samples and features: Random Patches

Scikit-Learn Bagging Methods

BaggingRegressor() Function

Parameters

- base_estimator (default
- n_estimators (default = 100)

Scikit-Learn Bagging Methods

RandomForestRegressor Function

Bagging

```
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier
bagging= RandomForestRegressor(max_features=12, random_state=2)  # by using all features, we implement bagging
bagging.fit(x_train, y_train)
bagging_pred= bagging.predict(x_test)
bagging_mse= mean_squared_error(bagging_pred, y_test)
bagging_mse
```

```
11.56611499209486
```

Random Forest

```
forest= RandomForestRegressor(max_features=6, random_state=2)  # Select only 6 features at each decision point
forest.fit(x_train, y_train)
forest_pred = forest.predict(x_test)
forest_mse = mean_squared_error(forest_pred, y_test)
forest_mse
```

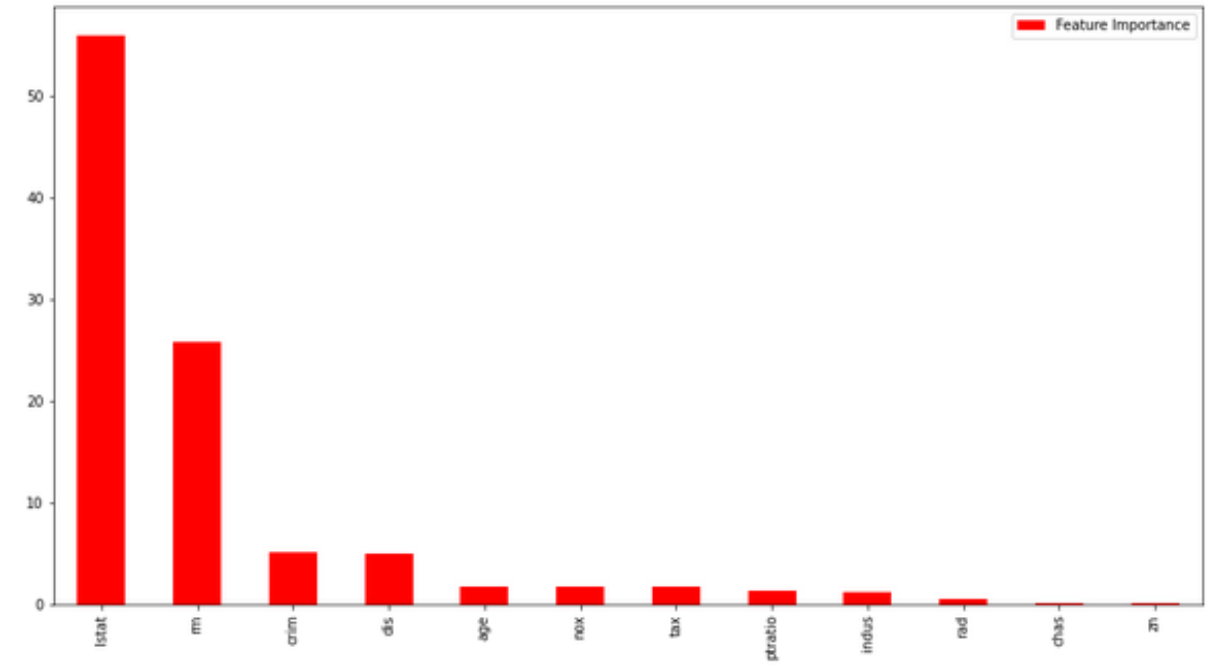
```
11.210173703557313
```

Variable Importance

Bagging

```
1 bagging_featureImportance = pd.DataFrame({'Feature Importance':bagging.feature_importances_*100},index =x.columns)
2 bagging_featureImportance.sort_values('Feature Importance', ascending=False).plot(kind='bar', color='red')
```

```
<AxesSubplot:>
```

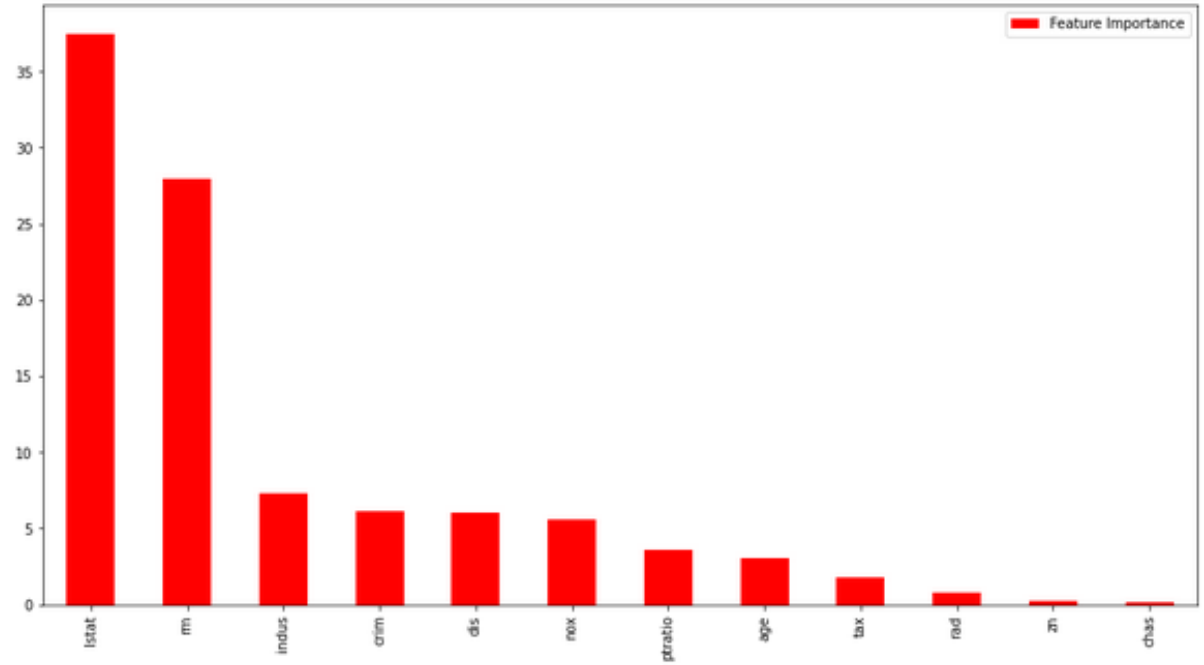


Variable Importance

Forest

```
1 rf_featureImportance= pd.DataFrame({'Feature Importance':forest.feature_importances_*100}, index= x.columns)
2 rf_featureImportance.sort_values('Feature Importance', ascending=False).plot(kind='bar', color='red')
```

```
<AxesSubplot:>
```



Boosting

Boosting

Overview

- A third type of ensemble modeling technique (after bagging and random forests) is ***boosting***
- Unlike the other techniques, this approach is a ***sequential*** ensemble method
  - The residual (or misclassified point) from the previous learner is incorporated in the next run of the algorithm

Boosting

Overview

Many boosting methods have been proposed, but the two most popular are:

- AdaBoost (Adaptive Boosting)
  - Iteratively adjusts the weights on “hard” observations in the purity calculations for deciding on the next split
- Gradient Boosting
  - Iteratively fits subsequent models to the residuals from the previous model and combines the models together

AdaBoost

Overview

- “Pays more attention” to the training observations that the predecessor model missed
  - New predictors focus more on the “hard cases”
- For example, for an AdaBoost classifier:
  - Trains a “base classifier” (e.g., decision tree) and uses it to make predictions on the training set
  - Increase the relative weights on the misclassified observations that are used to calculate the purities of the regions
  - Repeat

AdaBoost

Overview

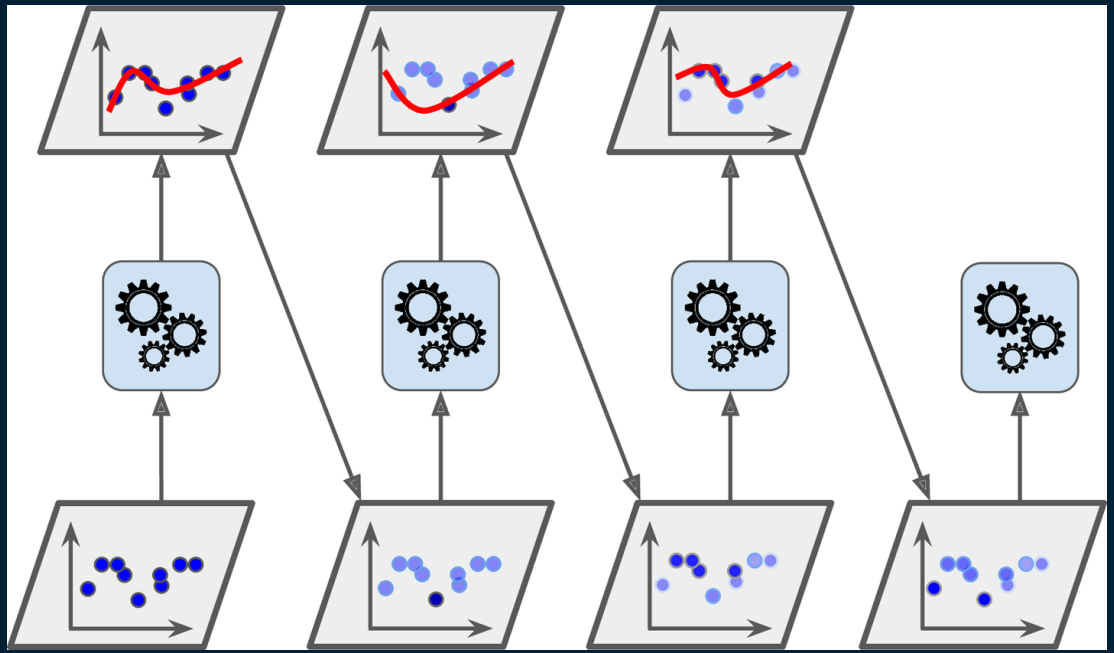


Figure from *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd edition, Aurelien Geron, O'Reilly Media

Gradient Boosting

Basic Approach

Given a set of data (x_1, y_1), (x_2, y_2), … (x_n, y_n) and an initial decision tree model F, calculate the residuals $y_n - F(x_n)$

- You want to add an additional model h to F so that the new prediction will be $F(x) + h(x)$ so that

$$\begin{aligned} F(x_1) + h(x_1) &= y_1 \\ F(x_2) + h(x_2) &= y_2 \\ &\vdots \\ F(x_n) + h(x_n) &= y_n \end{aligned}$$

Or, equivalently,

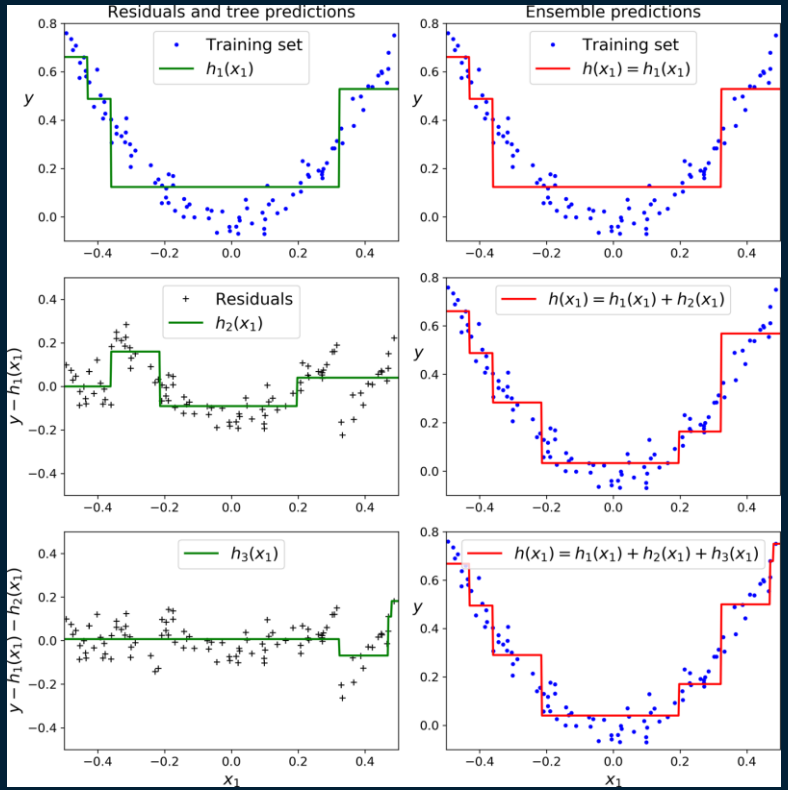
$$\begin{aligned} F(x_1) + h(x_1) &= y_1 \\ F(x_2) + h(x_2) &= y_2 \\ &\vdots \\ \underbrace{F(x_n) + h(x_n)}_{\text{Residuals}} &= y_n \end{aligned}$$

Gradient Boosting

First model fit

Fitting a model to the residuals from the first model

Fitting a model to the residuals from the second model



Combining the first two models to form the second

Combining the first three models to form the third

Figure from *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd edition, Aurelien Geron, O'Reilly Media

Gradient Boosting

Hyperparameters

- Number of trees B. Unlike bagging and random forests, overfitting is a possibility
- Shrinkage parameter λ (also referred to as tuning parameter)
- Number of splits d in each tree.

Advantages of Gradient Boosting

- Less data pre-processing (than neural networks, but same as forests) is required as trees automatically handle missing values and variable reduction.
- Often outperforms other classes of models, as boosting reduces the correlation of the predictions of the trees, which in turn improves the predictions of the boosting model.

Disadvantages of Gradient Boosting

- Training generally takes longer because trees are built sequentially.
- Gradient boosting models are more sensitive to extreme values and anomalies in the data.
- Can be slow for real-time scoring.