Classification and Regression Trees (CART)

Data Mining for Business Analytics in Python

Shmueli, Bruce, Gedeck & Patel

 rev 11/12/19

Trees and Rules

Goal: Classify or Predict an outcome based on a set of predictors

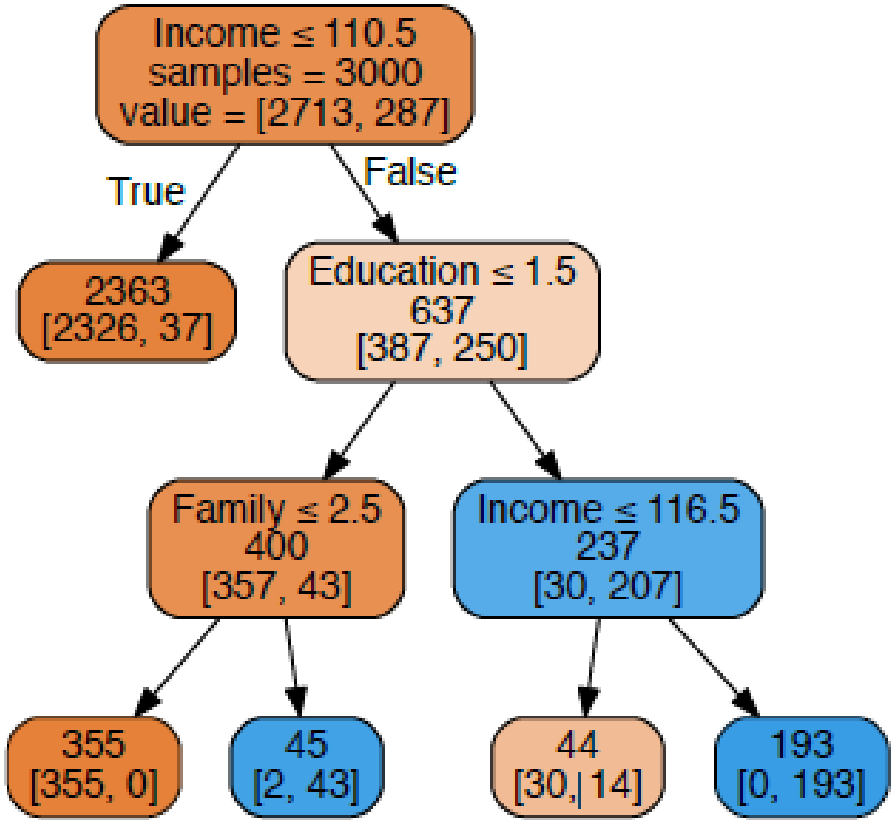
The output is a set of **rules**

Example:

- Goal: Classify a record as “will accept credit card offer” or “will not accept”
- Rule might be “IF (Income >= 106) AND (Education < 1.5) AND (Family <= 2.5) THEN Class = 0 (nonacceptor)
- Also called CART, Decision Trees, or just Trees
- Rules are represented by tree diagrams

Example Tree:

Classify Bank Customers as Loan Acceptors Y/N



Import Needed Functionality

```
import pandas as pd
import numpy as np
from sklearn.tree import DecisionTreeClassifier,
     DecisionTreeRegressor
from sklearn.ensemble import RandomForestClassifier,
     GradientBoostingClassifier
from sklearn.model_selection import train_test_split,
     cross_val_score, GridSearchCV
import matplotlib.pylab as plt
from dmba import plotDecisionTree,
     classificationSummary, regressionSummary
```

How Is the Tree Produced?

Two steps are taken:

1. **Recursive partitioning:** Repeatedly split the records into two parts to achieve maximum homogeneity of outcome within each new part

1. **Pruning (Stopping Tree Growth):** Cutting the tree back
   (A fully grown tree is too complex and will overfit)

Video-1

Video-2

Video-3

Tree Structure

Two types of nodes in a tree:

1. Decision (splitting) nodes: They have successors
2. Terminal nodes (leaves of the tree): Represent the partitioning of the data by predictors.

Recursive Partitioning

Recursive Partitioning Steps

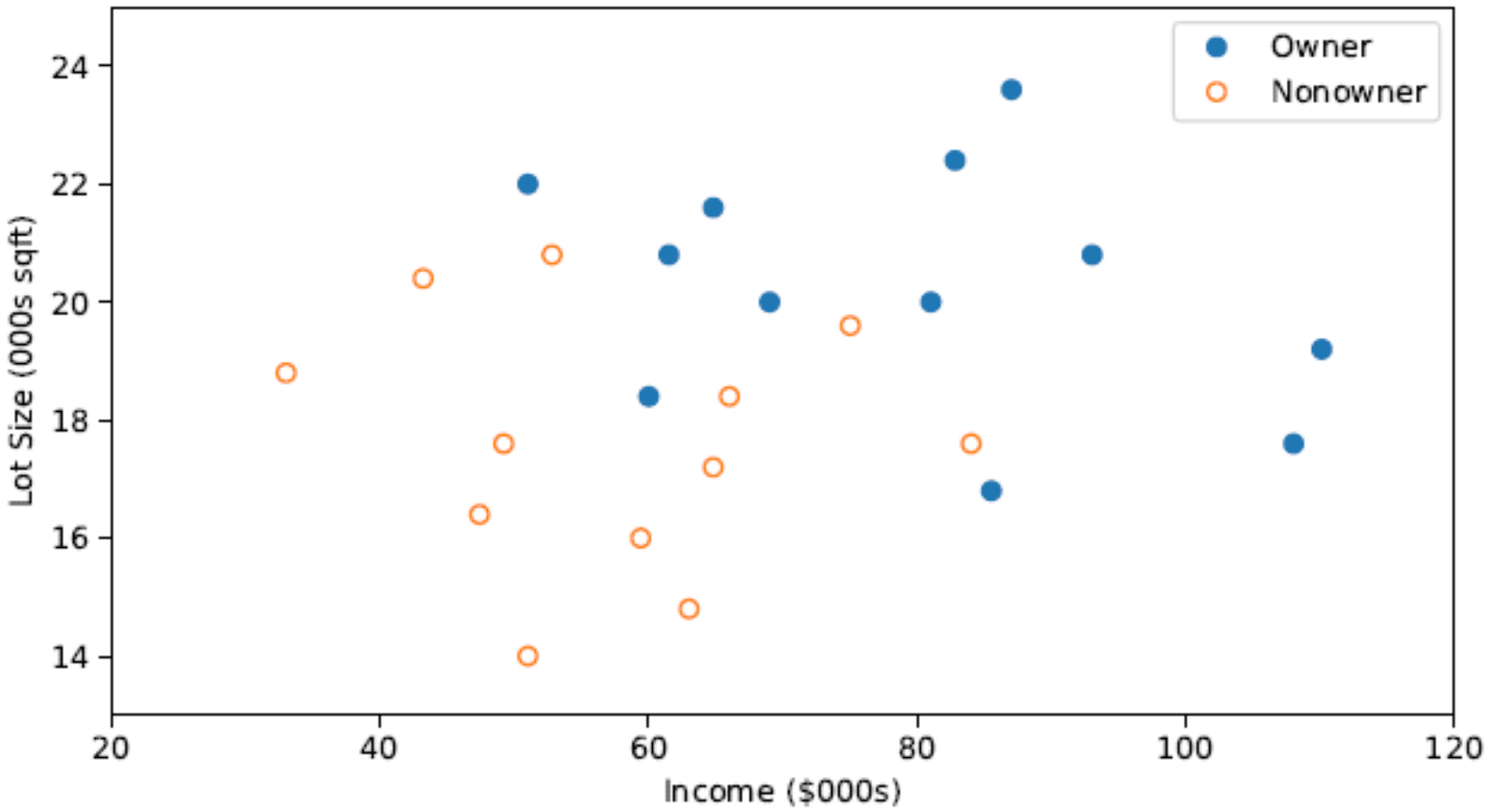
- Pick one of the predictor variables, x_i
- Pick a value of x_i, say s_i, that divides the training data into two (not necessarily equal) portions
- Measure how "pure" or homogeneous each of the resulting portions is

  "Pure" = containing records of mostly one class (or, for prediction, records with similar outcome values)
- Algorithm tries different values of x_i, and s_i to maximize purity in initial split
- After you get a "maximum purity" split, repeat the process for a second split (on any variable), and so on
- The idea is to find the split (subset) with the lowest MSE

Example: Riding Mowers

- Goal: Classify 24 households as owning or not owning riding mowers
- Predictors = Income, Lot Size

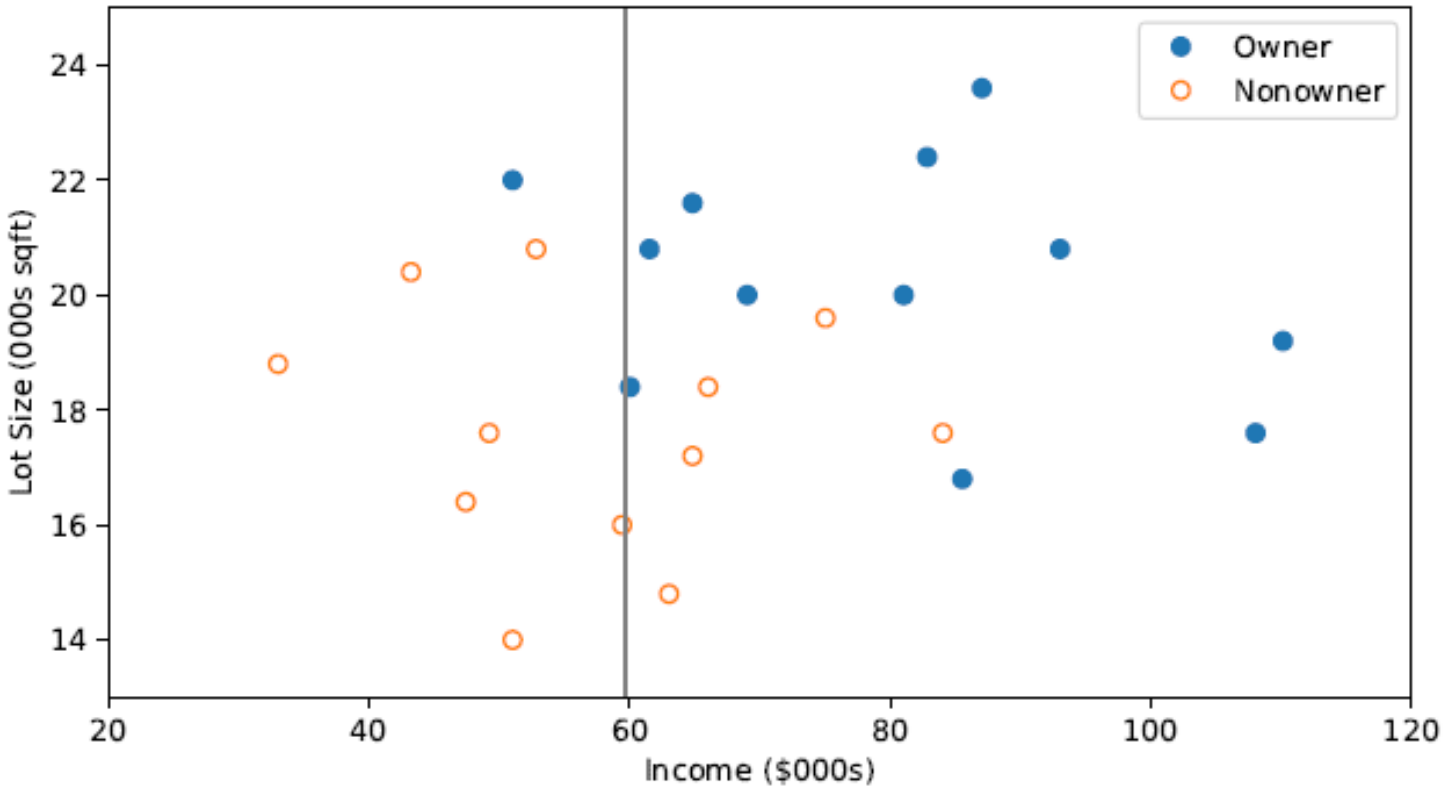
Income	Lot_Size	Ownership
60.0	18.4	owner
85.5	16.8	owner
64.8	21.6	owner
61.5	20.8	owner
87.0	23.6	owner
110.1	19.2	owner
108.0	17.6	owner
82.8	22.4	owner
69.0	20.0	owner
93.0	20.8	owner
51.0	22.0	owner
81.0	20.0	owner
75.0	19.6	non-owner
52.8	20.8	non-owner
64.8	17.2	non-owner
43.2	20.4	non-owner
84.0	17.6	non-owner
49.2	17.6	non-owner
59.4	16.0	non-owner
66.0	18.4	non-owner
47.4	16.4	non-owner
33.0	18.8	non-owner
51.0	14.0	non-owner
63.0	14.8	non-owner


Owner
Nonowner
Lot Size (000s sqft)
Income ($000s)
24
22
20
18
16
14
20
40
60
80
100
120

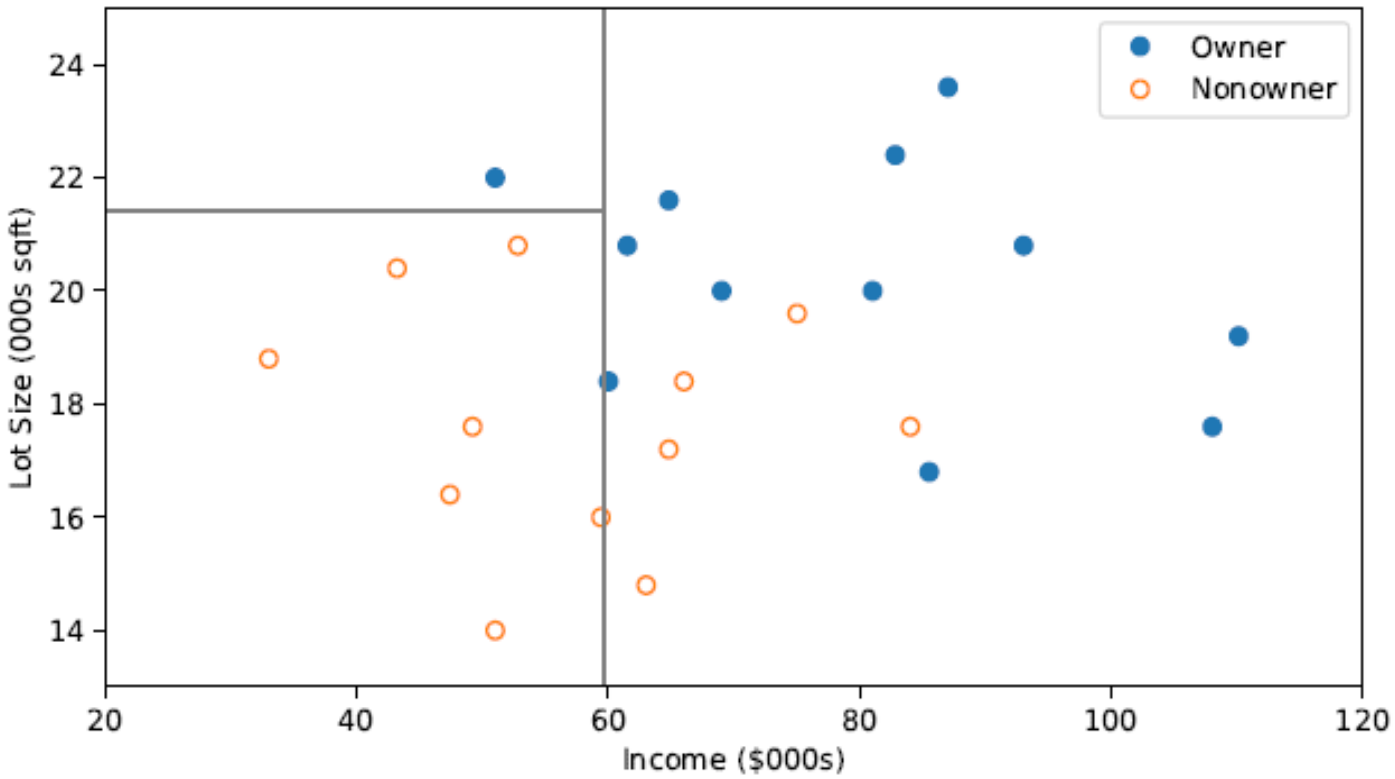
How to split

- Order records according to one variable, say income
- Take a predictor value, say 59.7 (the first record) and divide records into those with income >= 59.7 and those < 59.7
- Measure resulting purity (homogeneity) of class in each resulting portion
- Try all other split values
- Repeat for other variable(s)
- Select the one variable and split that yields the most purity

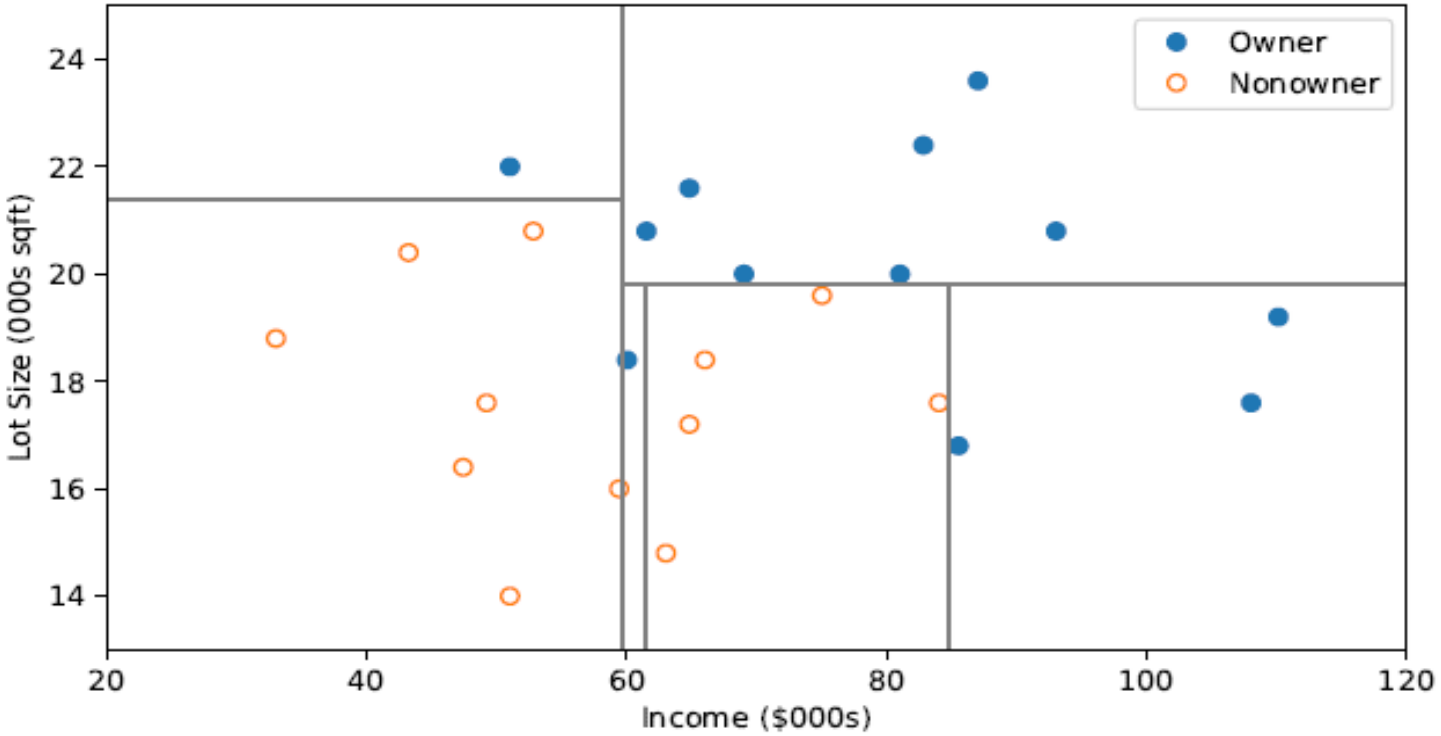
The first split: Income = 59.7



Second Split: Lot size = 21.4



After All Splits



- What are the possible split values for a variable?
  Simply the midpoints between pairs of consecutive values for each predictor.
- The split points are ranked according to how much they reduce impurity (heterogeneity) in the resulting rectangle.
- A pure rectangle is one that is composed of a single class.
- The reduction in impurity is defined as overall impurity before the split minus the sum of the impurities for the two rectangles that result from a split.

Note: Categorical Variables

- Examine all possible ways in which the categories can be split.
- E.g., categories A, B, C can be split in 3 ways

  {A} and {B, C}

  {B} and {A, C}

  {C} and {A, B}
- With many categories, # of splits becomes huge
- As with k-NN, a predictor with m categories (m>2) should be factored into m dummies (not m-1)

Normalization

- Whether predictors are numerical or categorical, it does not make any difference if they are standardized (normalized) or not.

Measuring Impurity

Gini Index

Gini Index for rectangle (leaf) *A*

$$I(A) = 1 - \sum_{k=1}^{m} p_k^2$$

p_k =Proportion of records in rectangle *A* that belong to class *k* (out of *m* classes)

- I(A) = 0 (Min) when all records belong to same class
- I(A) = (m-1)/m (Max) when all m classes are equally represented (= 0.50 in binary case)

Gini Index

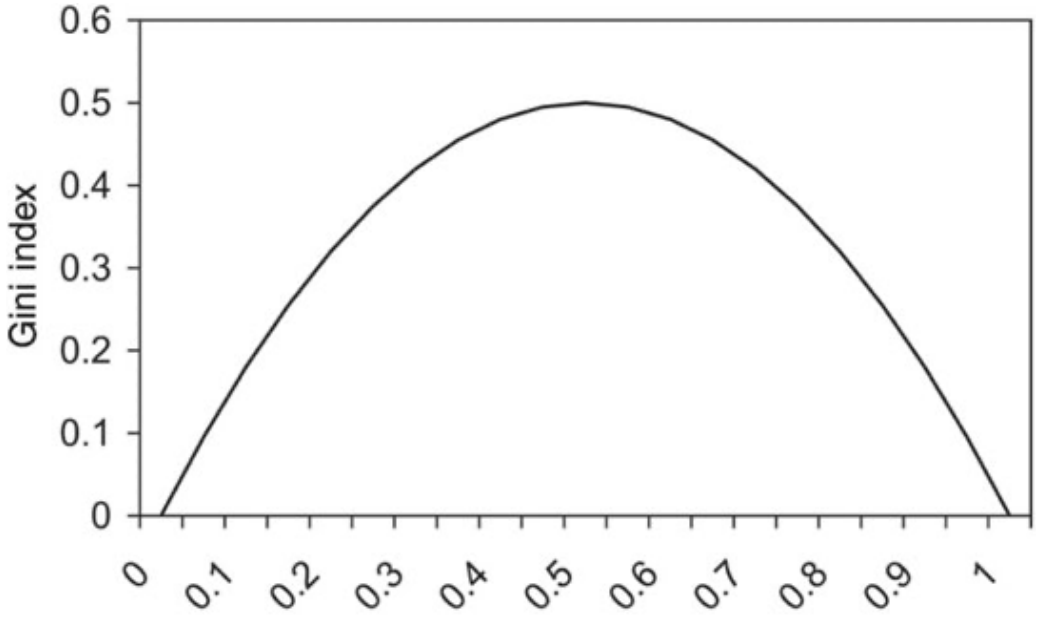
- Values of the Gini index for a two-class case as a function of p_k
- Impurity measure is at its peak when p_k = 0.5

  (when the rectangle contains 50% of each of the two classes)

- Example: A split made 80% of Class 1 and 20% of Class 2

  P_1 = 0.8 (80% Class 1) P_2 = 0.2 (20% Class 2).

- Now let's calculate the Gini index for this split:
- Gini = $1-(0.8^2+0.2^2) = 1-(0.64+0.04) = 1-0.68 = 0.32$



Entropy

$$entropy\,(A) = -\sum_{k=1}^{m} p_k \log_2(p_k)$$

p = Proportion of cases in rectangle A that belong to class k (out of m classes)

- entropy(A) = 0 (most pure, all records belong to the same class)
- entropy(A) = $\log_2(m)$ (equal representation of all m classes)
- In the two-class case, the entropy measure is maximized (like the Gini index) at p_k = 0.5.

Impurity and Recursive Partitioning

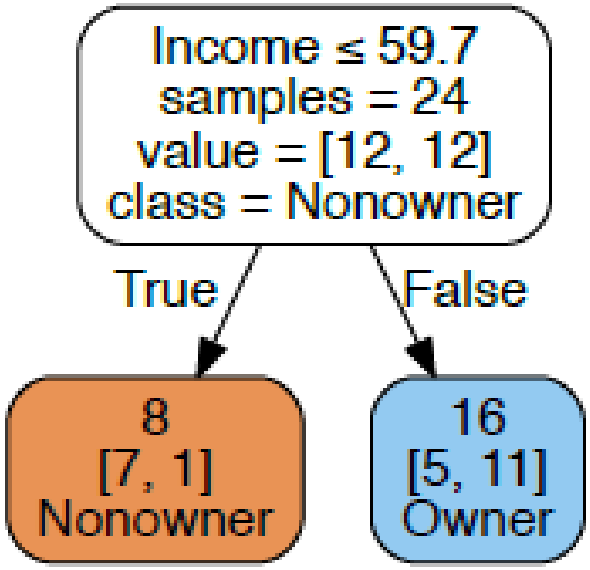
- Obtain overall impurity measure (weighted avg. of individual rectangles)
- At each successive stage, compare this measure across all possible splits in all variables
- Choose the split that reduces impurity the most
- Chosen split points become nodes on the tree

Running the Classification Tree

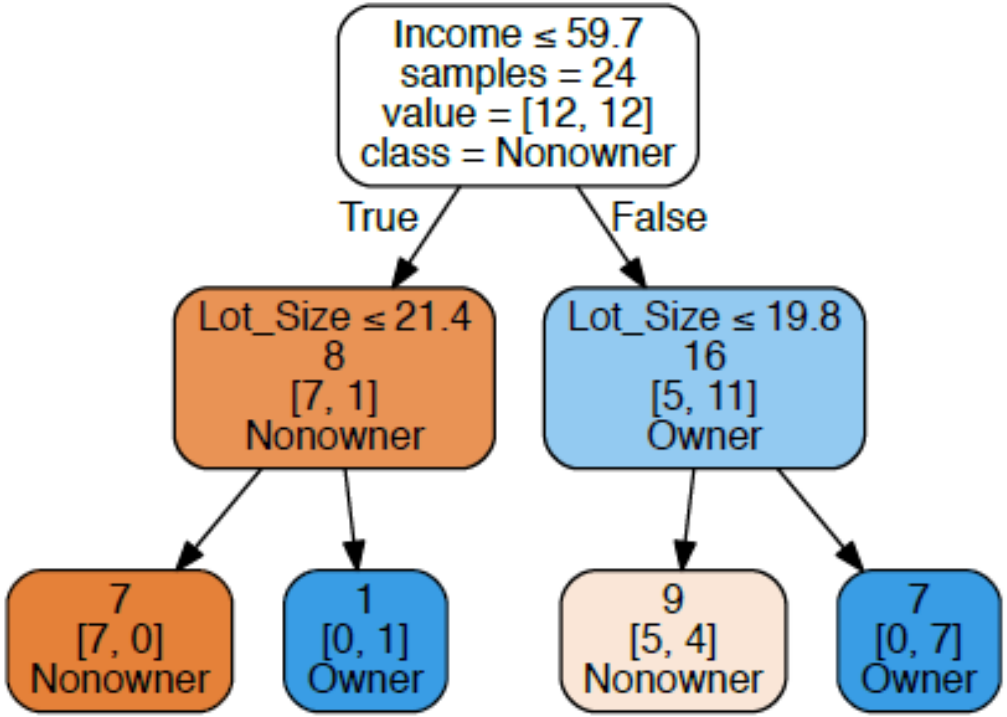
```
mower_df = pd.read_csv('RidingMowers.csv')
# use max_depth to control tree size (None = full tree)
classTree = DecisionTreeClassifier(random_state=0, max_depth=1)
classTree.fit(mower_df.drop(columns=['Ownership']),
   mower_df['Ownership'])
print("Classes: {}".format(', '.join(classTree.classes_)))
plotDecisionTree(classTree,feature_names=mower_df.columns[:2],
   class_names=classTree.classes_)
```

- Typical values for max_depth range from 1 (a very shallow tree) to a large positive integer, depending on the complexity of your data and the trade-off between underfitting and overfitting.
- classTree.classes_: This is an attribute of the DecisionTreeClassifier that contains the unique class labels the classifier has learned. It is an array or list of class labels.

First Split – The Tree

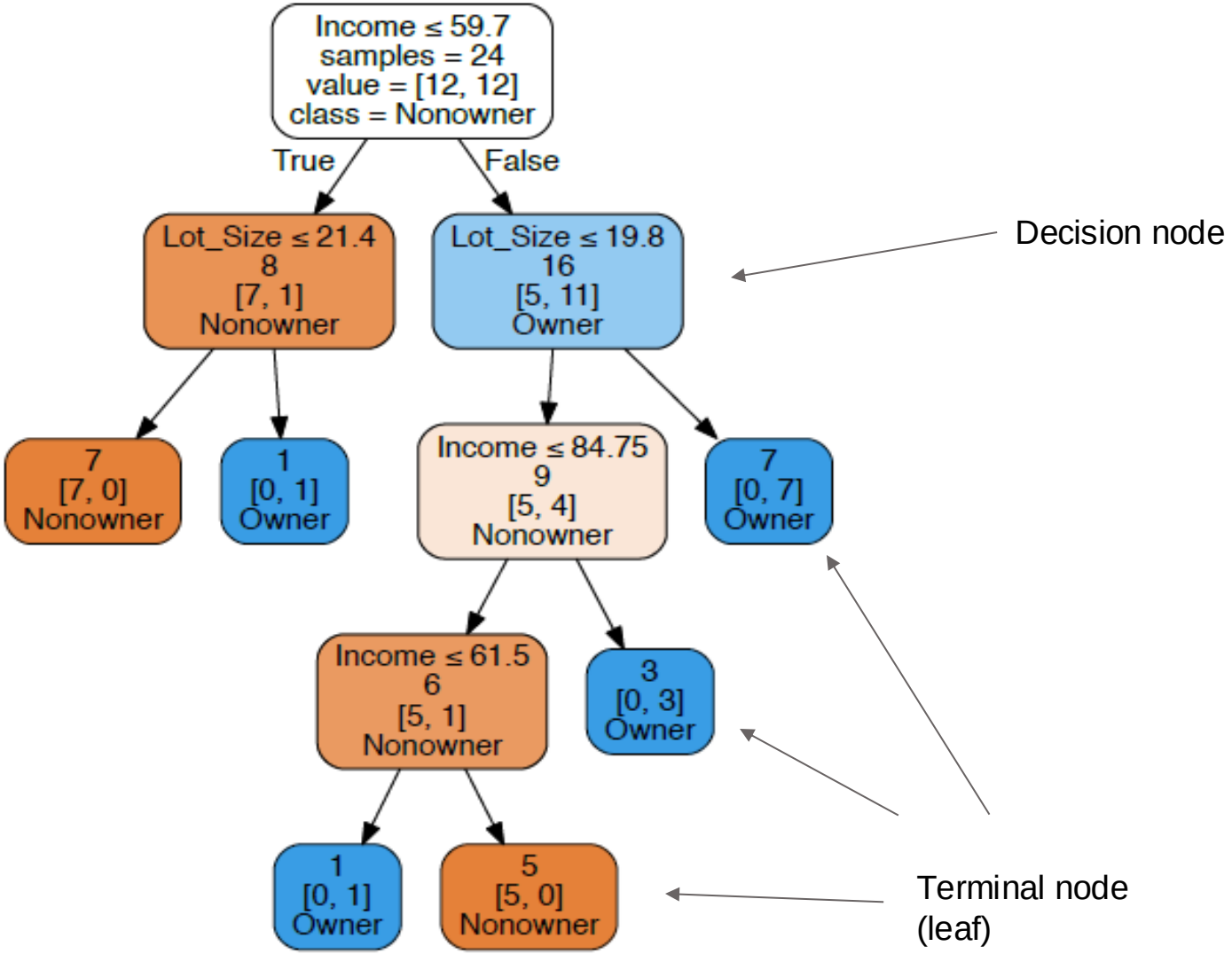


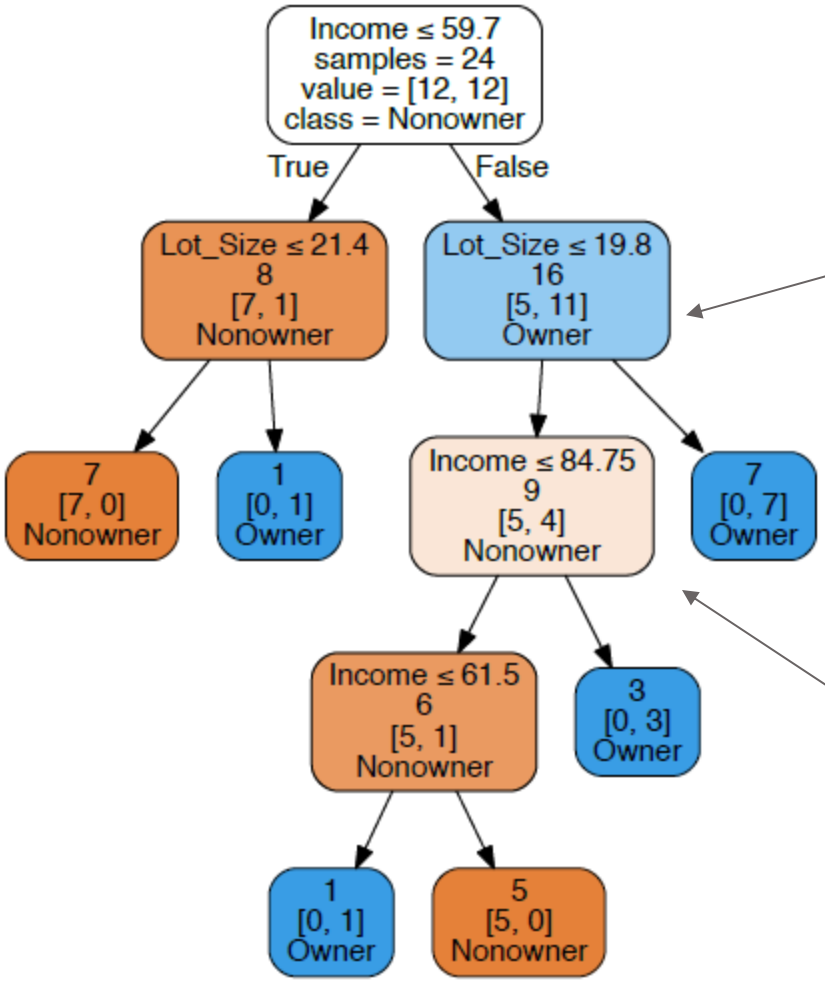
Tree After 2 Splits



The first split is on Income, then the next split is on Lot Size for both the low income group (at lot size 21.4) and the high income split (at lot size 19.8)

Tree After All Splits



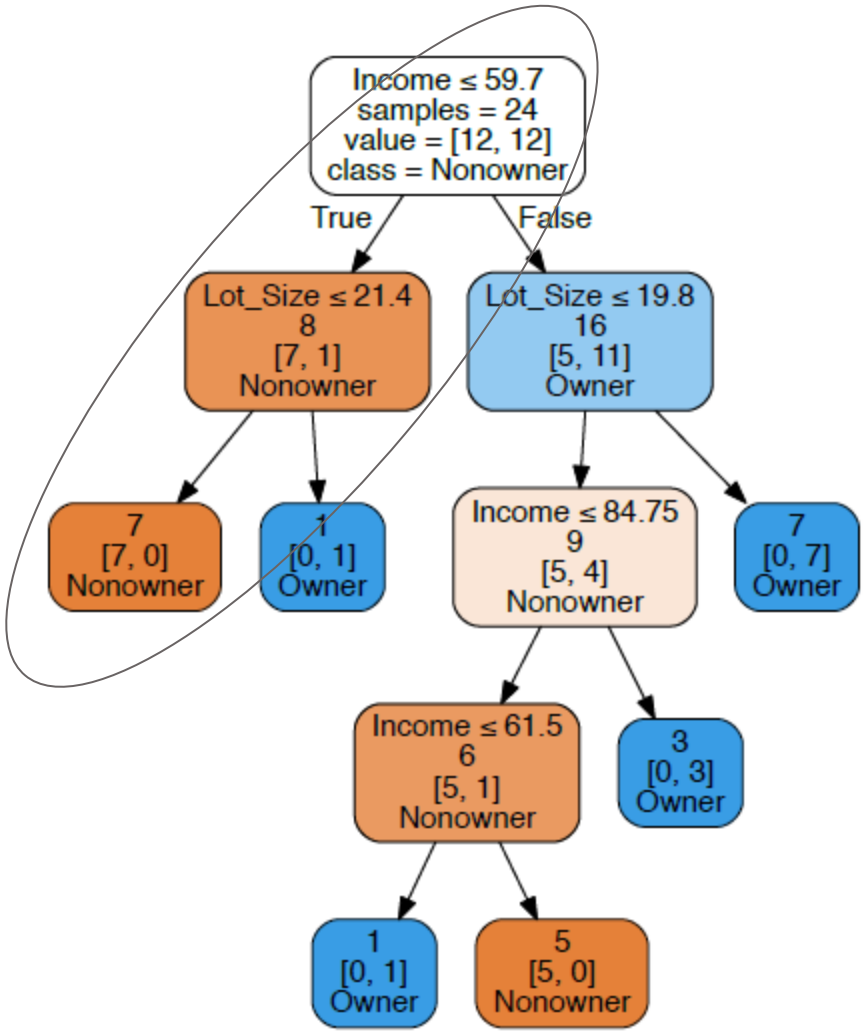


The dominant class in this portion of the first split (those with income >= 59.7) is “owner” with 11 owners and 5 non-owners

The next split for this group of 9 will be on the basis of Income, splitting at 84.75

Read down the tree to derive rules

If Income <= 59.7 AND
Lot Size <= 21.4,
classify as “Nonowner”



Evaluating the Performance of a Classification Tree

- After fitting a model to training data, we always need to assess and tune the model, which is particularly important with classification and regression trees, for two reasons:

1. Tree structure can be quite unstable, shifting substantially depending on the sample chosen.

   Imagine that we partition the data randomly into two samples, A and B, and we build a tree with each. If there are several predictors of roughly equal predictive power, you can see that it would be easy for samples A and B to select different predictors for the top-level split, just based on which records ended up in which sample. And a different split at the top level would likely cascade down and yield completely different sets of rules. Therefore, we should view the results of a single tree with some caution.

2. A fully-fit tree will invariably lead to overfitting.

The Instability Problem

Trees can be unstable

- If 2 or more variables are of roughly equal importance, which one CART chooses for the first split can depend on the initial partition into training and validation
- A different partition into training/validation could lead to a different initial split
- This can cascade down and produce a very different tree from the first training/validation partition
- Solution is to try many different training/validation splits – “cross validation”

The Overfitting Problem

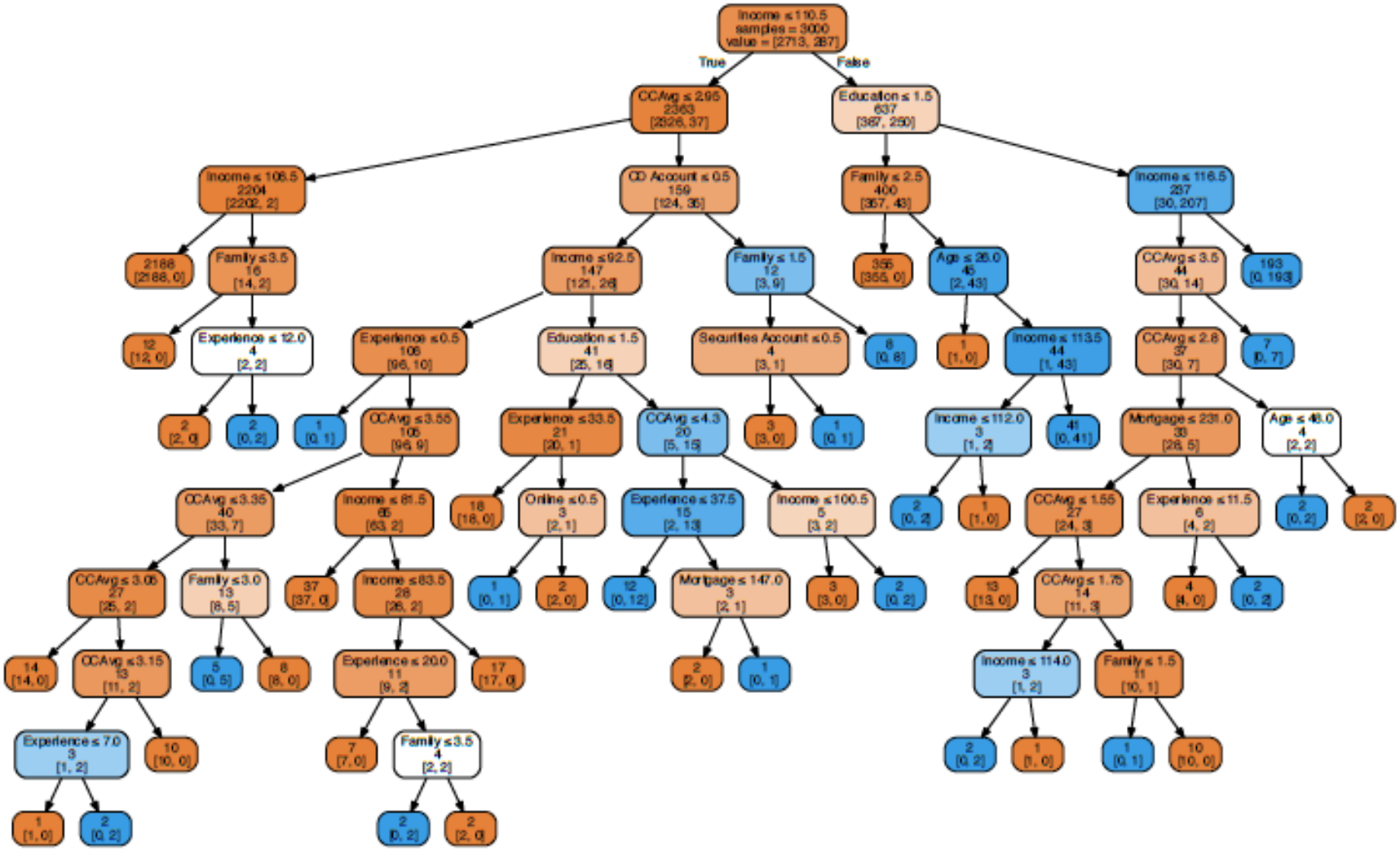
Full trees are complex and overfit the data

- Natural end of process is 100% purity in each leaf
- This **overfits** the data, which ends up fitting noise in the data
- Consider Example 2, Loan Acceptance with more records and more variables than the Riding Mower data – the full tree is very complex
- One intuitive reason a large tree may overfit is that its final splits are based on very small number of records. (In such cases, class difference is likely to be attributed to noise rather than predictor information)

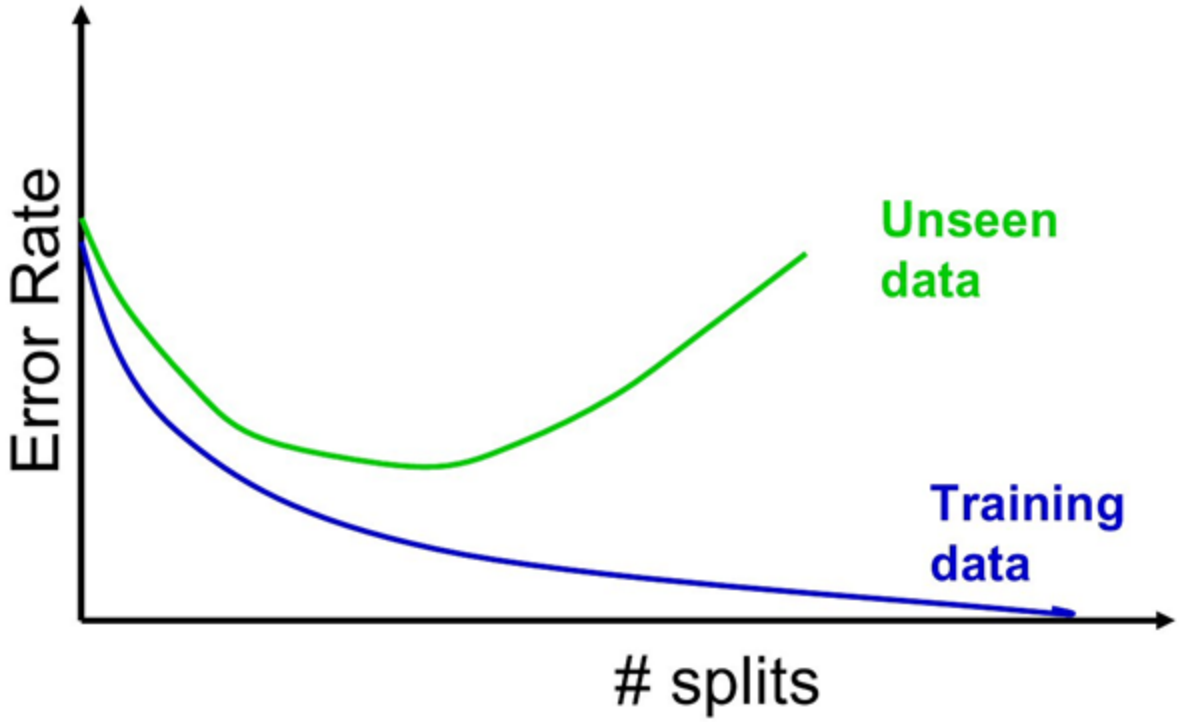
Example 2: Loan Acceptance

```
import pandas as pd
import numpy as np
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
import matplotlib.pylab as plt
from dmba import plotDecisionTree, classificationSummary, regressionSummary

bank_df = pd.read_csv('UniversalBank.csv')

bank_df.drop(columns=['ID', 'ZIP Code'], inplace=True)

X = bank_df.drop(columns=['Personal Loan'])
y = bank_df['Personal Loan']
train_X, valid_X, train_y, valid_y = train_test_split(X, y, test_size=0.4, random_state=1)

fullClassTree = DecisionTreeClassifier(random_state=1)
fullClassTree.fit(train_X, train_y)

plotDecisionTree(fullClassTree, feature_names=train_X.columns)
```

Full trees are too complex – they end up fitting noise, overfitting the data



Overfitting and instability produce poor predictive performance – past a certain point in tree complexity, the error rate on new data starts to increase



Assessing Tree Performance

```
classificationSummary(train_y, FullClassTree.predict(train_X))
classificationSummary(valid_y, fullClassTree.predict(valid_X))

Output
# full tree: training
Confusion Matrix (Accuracy 1.0000)
               Prediction
   Actual         0    1
   0           2727    0
   1              0  273

# full tree: validation
Confusion Matrix (Accuracy 0.9790)
               Prediction
   Actual         0    1
   0           1790   17
   1             25  168
```

Limiting Tree Growth

In `DecisionTreeClassifier()` we can control

- Tree depth
- Minimum number of records in split nodes
- Minimum number of records in terminal node
- Minimum impurity increase needed

But what are optimal values for these parameters?

With cross validation (CV), test multiple trees

- Partition data into sets (folds) for model-fitting, and data for evaluating
- Fit model with training fold, and evaluate with holdout fold
- Repeat, typically with evaluation folds that are non-overlapping and smaller than training folds
- e.g. 5-fold CV fits 5 models, each evaluated on 20% of the data that is held out, with each set of evaluation data non-overlapping with the others

With cross validation (CV), test multiple trees

```
treeClassifier = DecisionTreeClassifier(random_state=1)

scores = cross_val_score(treeClassifier, train_X, train_y, cv=5)

print('Accuracy scores for each fold: ', [f'{acc:.3f}' for acc
in scores])
```

```
Accuracy scores for each fold: ['0.988', '0.973', '0.993', '0.982', '0.993']
```

Exhaustive `GridsearchCV` For Most Accurate Parameters

Use `GridsearchCV` on training data, coupled with cross-validation.

- Exhaustively build trees using different values for the parameters, using the training fold
- Measure accuracy on the holdout fold
- Choose tree with best performance
- Assess its performance on the validation data, not used at all to this point

`RandomizedSearchCV()` as alternative to exhaustive search

`GridsearchCV` can be prohibitively time-consuming if many parameters are involved.

- `Randomized Search(CV)` is alternative
- Randomly selects from set of possible parameter combinations
- Use `n_iter` to set the number of permitted iterations

GridSearchCV() code

```
# Start with an initial guess for parameters
param_grid = {
  'max_depth': [10, 20, 30, 40],
  'min_samples_split': [20, 40, 60, 80, 100],
  'min_impurity_decrease': [0, 0.0005, 0.001, 0.005, 0.01]
}
# Which values are best?
# n_jobs=-1 will utilize all available CPUs
gridSearch =
GridSearchCV(DecisionTreeClassifier(random_state=1),
   param_grid, cv=5, n_jobs=-1)
gridSearch.fit(train_X, train_y)
print('Initial score: ', gridSearch.best_score_)
print('Initial parameters: ', gridSearch.best_params_)
```

GridSearchCV() code - refined

```
# Adapt grid based on result from initial grid search
param_grid = {
'max_depth': list(range(2, 16)), # 14 values
'min_samples_split': list(range(10, 22)), # 11 values
'min_impurity_decrease': [0.0009, 0.001, 0.0011], # 3 values
}
gridSearch =
GridSearchCV(DecisionTreeClassifier(random_state=1),
    param_grid, cv=5, n_jobs=-1)
gridSearch.fit(train_X, train_y)
print('Improved score: ', gridSearch.best_score_)
print('Improved parameters: ', gridSearch.best_params_)
bestClassTree = gridSearch.best_estimator_
```

Performance

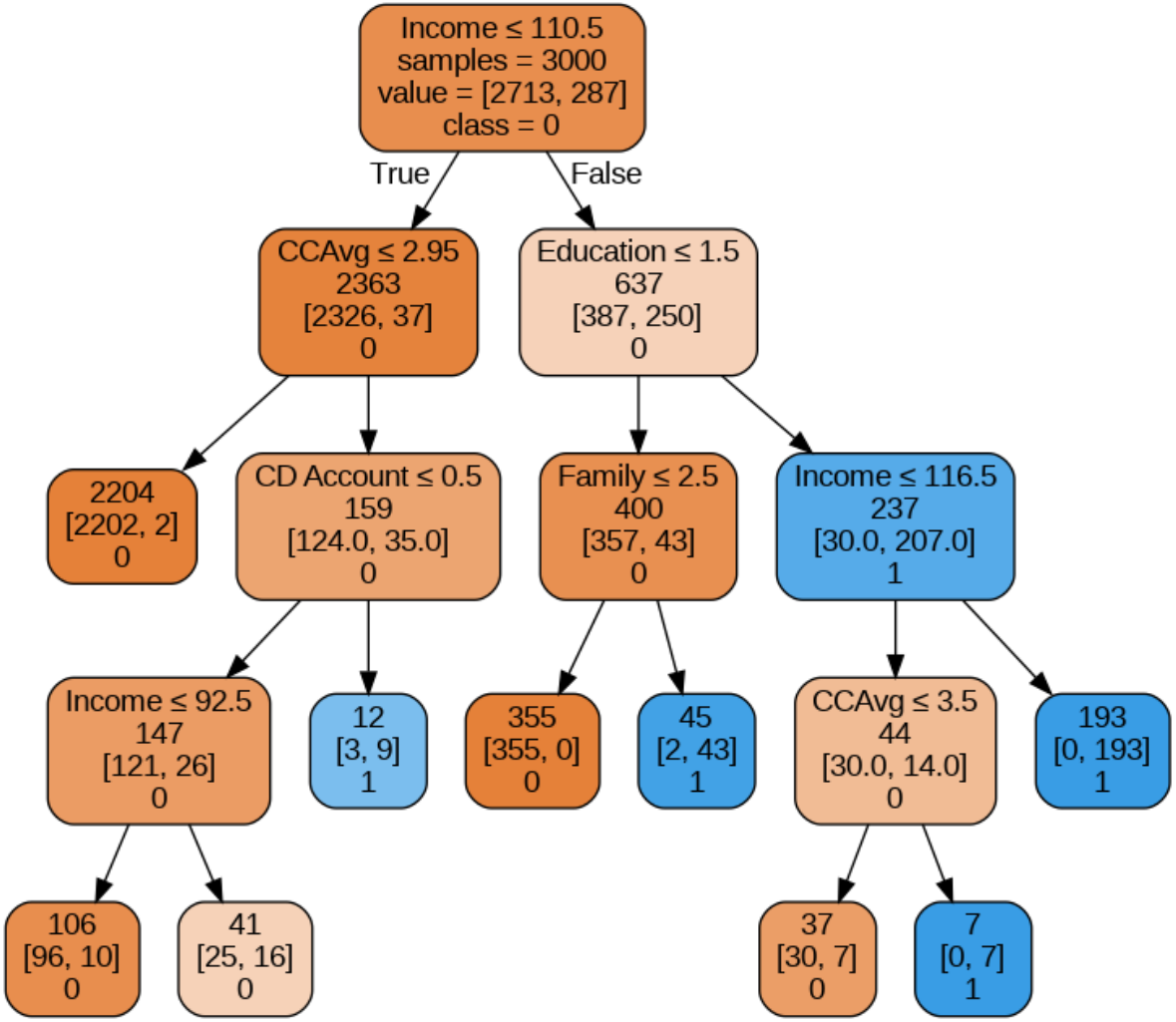
```
# Results from initial guess at parameters
Initial score: 0.9877
Initial parameters: {'max_depth': 10,
'min_impurity_decrease': 0.0005,
'min_samples_split': 20}
```

```
# Results after second Gridsearch
Improved score: 0.9873
Improved parameters: {'max_depth': 4,
'min_impurity_decrease': 0.0011,
'min_samples_split': 13}
```

GridSearchCV() results

```
# Recall original full tree
# full tree: validation
Confusion Matrix (Accuracy 0.9790)
              Prediction
   Actual        0    1
   0          1790   17
   1            25  168


# Results after Gridsearch
# full tree: validation
Confusion Matrix (Accuracy 0.9815)
              Prediction
   Actual        0    1
   0          1801    6
   1            31  162
```

Resulting Tree



Other Methods for Limiting Tree Size

- CHAID (Chi-squared Automatic Interaction Detection)
  - ✓ Widely used in database marketing
  - ✓ Uses a well-known statistical test (the chi-square test for independence) to assess whether splitting a node improves the purity by a statistically significant amount.
- Pruning
  - ✓ Recognize that a very large tree is likely to overfit the training data, and that the smallest branches, which are the last to be grown and are furthest from the trunk, are likely fitting noise in the training data.

Regression Trees

Regression Tree is Similar, Except...

- Prediction is computed as the **average** of numerical target variable in the rectangle (in CT it is majority vote)
- Impurity measured by **sum of squared deviations** from leaf mean
- Performance measured by RMSE (root mean squared error)

Ensembles and Variants

- Better performance can be provided by several extension to trees that combine results from multiple trees.

  - Random Forest
  - Boosted Trees
  - Harness “wisdom of the crowd”

Random Forest

- A special case of bagging, a method for improving predictive power by combining multiple classifiers or prediction algorithms
- Draw multiple random samples, with replacement, from data ("bootstrap resampling")
- Fit (classification or regression) tree to each resample using a *random set of predictors*
- Combine the classifications/predictions from all the resampled trees (the "forest") to obtain improved predictions. Use voting for classification and averaging for prediction
- Basic idea: Taking an average of multiple estimates (models) is more reliable than just using a single estimate

Random Forest

```
bank_df = pd.read_csv('UniversalBank.csv')
bank_df.drop(columns=['ID', 'ZIP Code'], inplace=True)


X = bank_df.drop(columns=['Personal Loan'])
y = bank_df['Personal Loan']
train_X, valid_X, train_y, valid_y = train_test_split(X, y,
   test_size=0.4, random_state=1)
rf = RandomForestClassifier(n_estimators=500, random_state=1)
# n_estimators= Number of trees in the forest, Default = 100
rf.fit(train_X, train_y)
```

- criterion: {"gini", "entropy", "log_loss"}, default="gini" . The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "log_loss" and "entropy" both for the Shannon information gain.
- max_depth: default=None. The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
- min_samples_split: Min number of samples needed to continue splitting. default=2.

Variable Importance

- Results from a random forest cannot be displayed in a tree-like diagram
- Each variable is used by some trees and not others
- We can, therefore, measure each variable's contribution to reducing impurity (summing up the reduction in Gini Index for that predictor over all the trees in the forest)
- This is the *variable importance score*

Variable Importance, code

```
# variable (feature) importance plot
importances = rf.feature_importances_
std = np.std([tree.feature_importances_ for tree in
   rf.estimators_], axis=0)
df = pd.DataFrame({'feature': train_X.columns,
   'importance': importances, 'std': std})
df = df.sort_values('importance')
print(df)
ax = df.plot(kind='barh', xerr='std', x='feature',
   legend=False) # Error bars help communicate the uncertainty associated with the
importance scores. A wider error bar suggests more variability or uncertainty in the estimate.
ax.set_ylabel('')
plt.show()
```

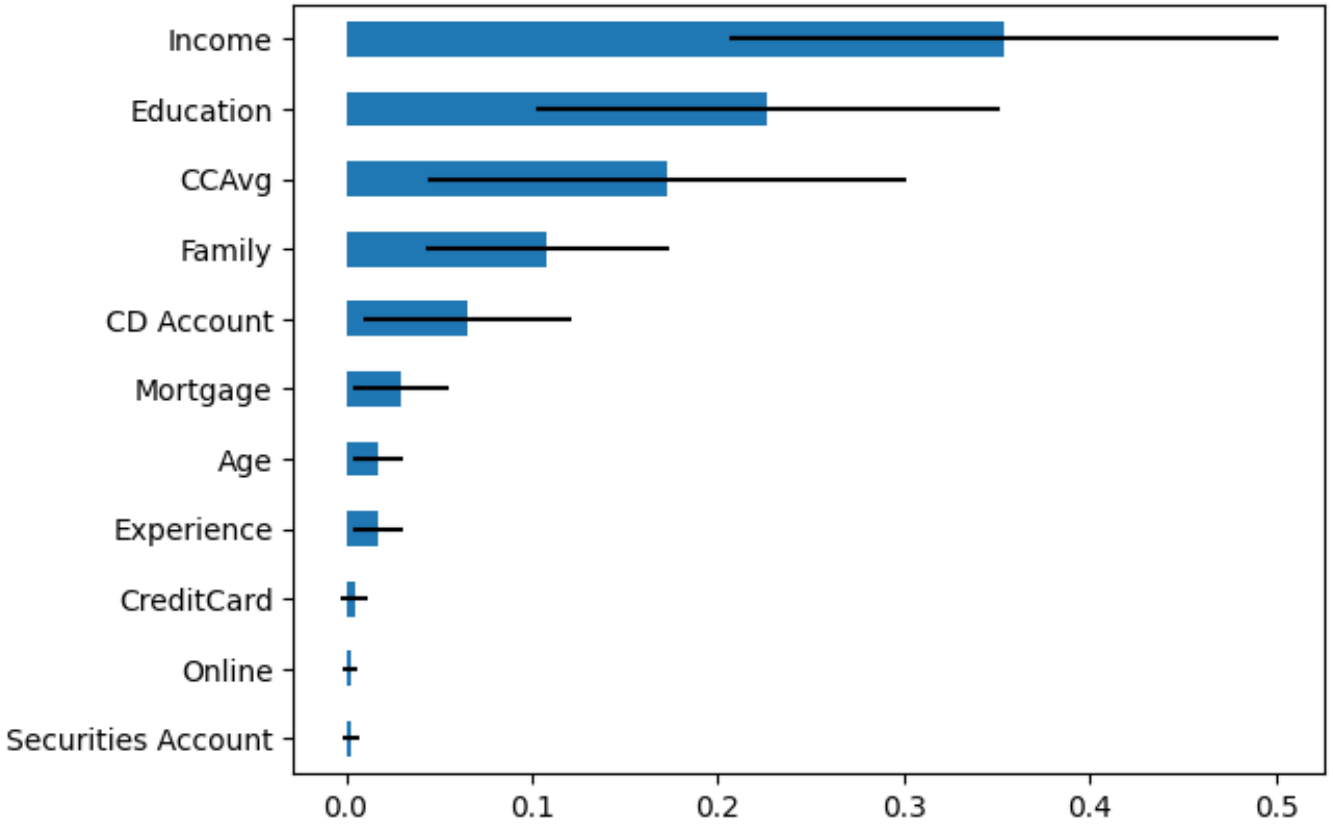
- The importance score for a particular predictor is computed by summing up the decrease in the Gini index for that predictor over all the trees in the forest.
- kind='barh': This specifies that you want to create a horizontal bar chart.
- The axis=0 argument indicates that the standard deviation should be calculated along the columns, effectively giving you the standard deviation for each feature.
- The 'xerr' parameter is used to plot error bars on the horizontal bars.
- Error bars provide insights into the stability of importance scores across different model variations.

Variable Importance scores

and their variability across trees

	feature	importance	std
7	Securities Account	0.002299	0.004525
9	Online	0.002310	0.004339
10	CreditCard	0.004489	0.007444
1	Experience	0.017199	0.013386
0	Age	0.017205	0.013555
6	Mortgage	0.029679	0.025951
8	CD Account	0.065396	0.055993
3	Family	0.108317	0.065242
4	CCAvg	0.172926	0.128754
5	Education	0.226659	0.124411
2	Income	0.353520	0.147712

Variable Importance Plot



Boosted Trees

- Fits a succession of single trees
- Each tree concentrates on misclassified records from the previous tree. Each successive fit up-weights the misclassified records from prior stage
- You now have a set of classifications or predictions, one from each tree
- Use weighted voting for classification, weighted average for prediction, higher weights to later trees

Boosted Trees

- Especially useful for the “rare case” scenario (suppose 1’s are the rare class)
- With simple classifiers, it can be hard for a “1” to “break out” from the dominant classification, & many get misclassified
- Up-weighting them focuses the tree fitting on the 1’s, and reduces the dominating effect of the 0’s

Boosted Trees - Performance

```
boost = GradientBoostingClassifier()
# n_estimators default=100

boost.fit(train_X, train_y)
classificationSummary(valid_y, boost.predict(valid_X))
```

```
Output
Confusion Matrix (Accuracy 0.9835)
            Prediction
Actual         0       1
0           1799       8
1             25     168
```

Advantages and Disadvantage of trees

- Easy to use and understand
- Single trees produce rules that are easy to interpret and implement
- Variable selection and reduction is automatic
- Do not require the assumptions of statistical models
- Can work without extensive handling of missing data
- Disadvantage of single trees: Instability and poor predictive performance