Introduction to Tree-Based Methods

Zachary Alexander
Galvanize
zachary.alexander@galvanize.com

Goals of this workshop

- Have a working *theoretical* knowledge of decision trees for regression, classification, and other problems.
- Understand the drawbacks of decision trees and how various ensemble methods (bagging, random forest, boosting) address these problems.
- Understand how to interpret ensemble tree-based methods.
- Use Python and Scikit-Learn to implement, visualize, and interpret tree-based models.

GitHub

Slides and iPython Notebooks available here:

https://github.com/zachalexan/tree-based-methods-workshop

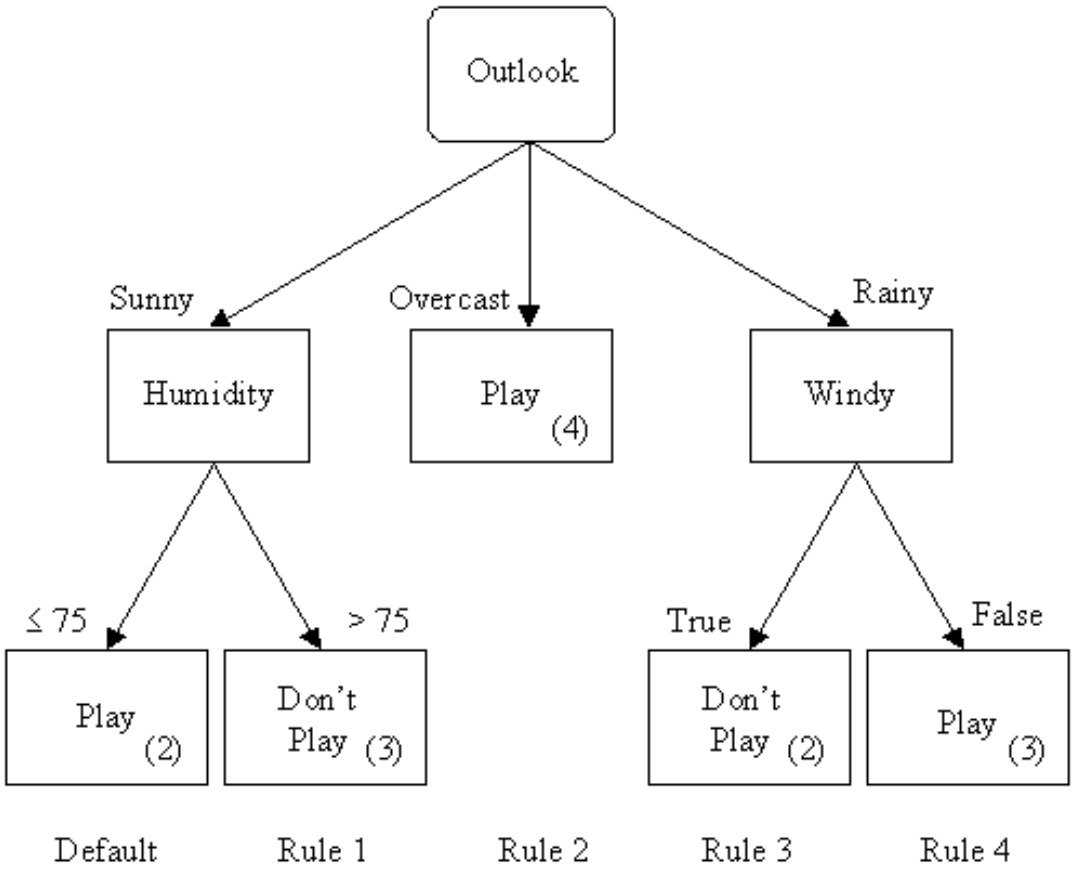
Experimental Code

- At the end of this talk, there are some *experimental* examples
- The relevant code-snippets come from DecisionPaths.ipynb
- They are dependent on a personal fork of sklearn:
  - https://github.com/andosa/scikit-learn/tree/tree_paths
  - http://blog.datadive.net/interpreting-random-forests/
- The best way I know of to try it out:
  - Uninstall sklearn (conda uninstall sklearn)
  - Clone the repo above
  - Run python setup.py install

Overview

- **Decision Trees**
  - Classification
  - Regression
- **Ensembles of Decision Trees**
  - Bagging / Random Forest
  - Gradient Boosting
- **Model Interpretation**
  - Feature Importance for ensembles of trees
  - Calculating Decision Paths

Decision Trees



A decision tree is just a hierarchy of rules that leads to a decision

Iris Data

- Four measurements for each flower.
- Goal is to classify each flower as one of 3 species

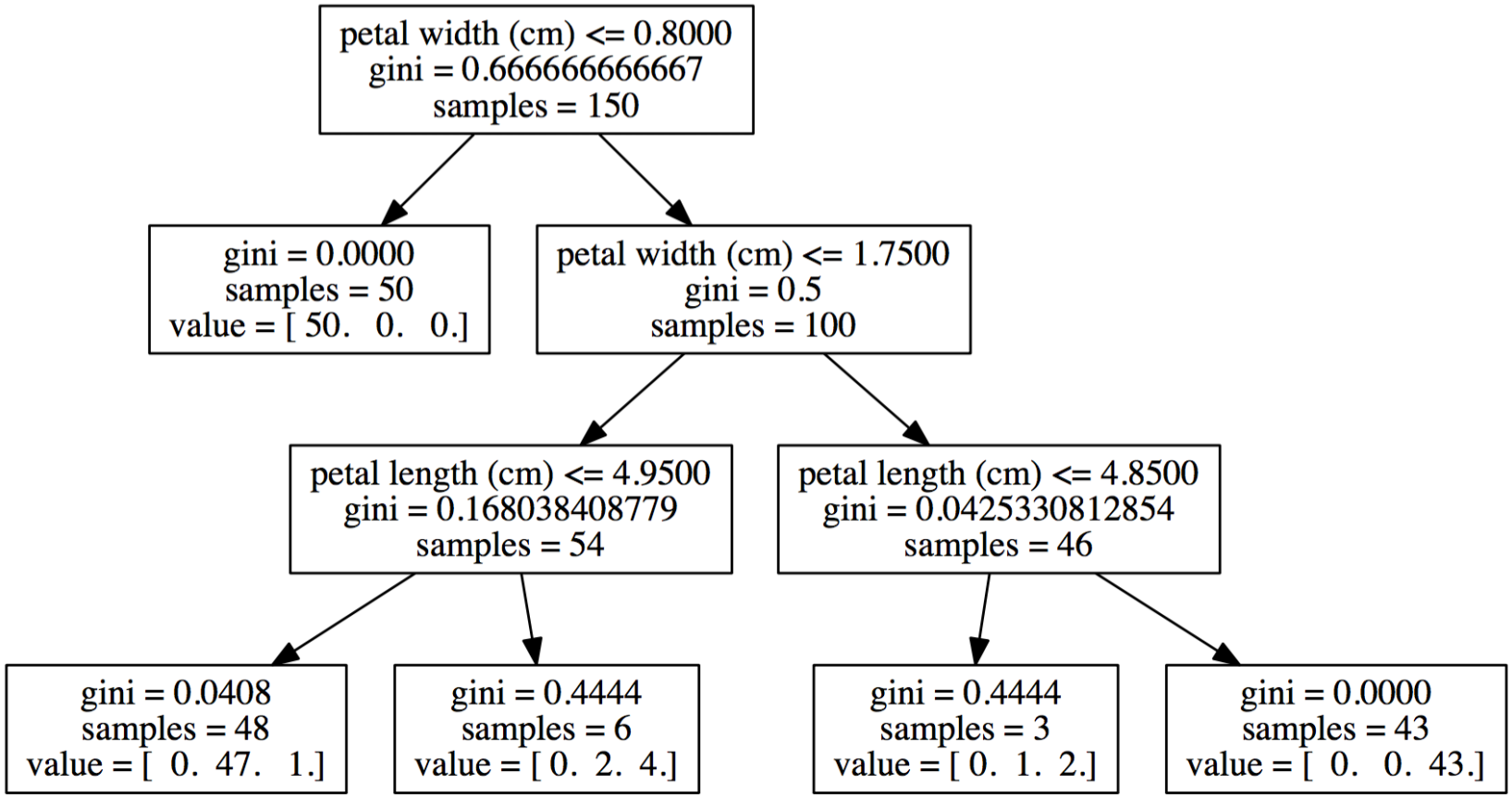
Sample Records

sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	Species
7.2	3.6	6.1	2.5	2
5.1	3.7	1.5	0.4	0
6.1	3.0	4.9	1.8	2
4.9	2.5	4.5	1.7	2
5.6	2.9	3.6	1.3	1

Data Description

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	Species
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667	1.000000
std	0.828066	0.433594	1.764420	0.763161	0.819232
min	4.300000	2.000000	1.000000	0.100000	0.000000
25%	5.100000	2.800000	1.600000	0.300000	0.000000
50%	5.800000	3.000000	4.350000	1.300000	1.000000
75%	6.400000	3.300000	5.100000	1.800000	2.000000
max	7.900000	4.400000	6.900000	2.500000	2.000000

Iris Decision Tree



Building a Decision Tree

Pseudocode

```
function BuildTree:
    If every item in the dataset is in the same class
    or there is no feature left to split the data:
        return a leaf node with the class label
    Else:
        find the best feature and value to split the data
        split the dataset
        create a node
        for each split
            call BuildTree and add the result as a child of
            the node
        return node
```

Basically, you just recursively split the data into two groups *in the best possible way*

Splitting Criterion for Classification

Two common optimization functions:

Information Gain

- ➡ Based on the concept of *Entropy* from information theory

Gini Impurity

- ➡ Based on the probability of ‘guessing wrong’

There is little difference in the outcome between these methods - mostly a matter of taste.

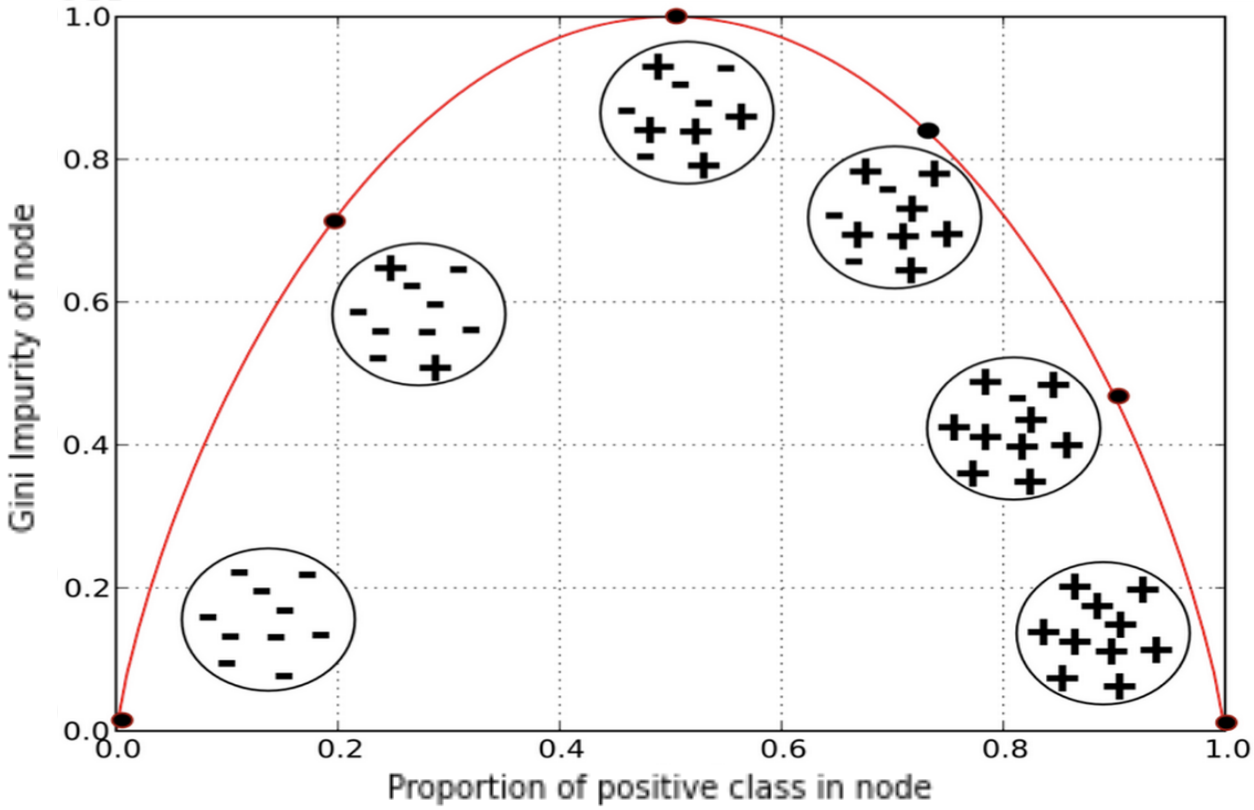
Gini Impurity

Gini impurity is one way of measuring which split in a decision tree is best. It is a measure of the following probability:

1. Take a random element from the set
2. Label it randomly according to the distribution of labels in the set
3. What is the probability that it is labeled incorrectly?

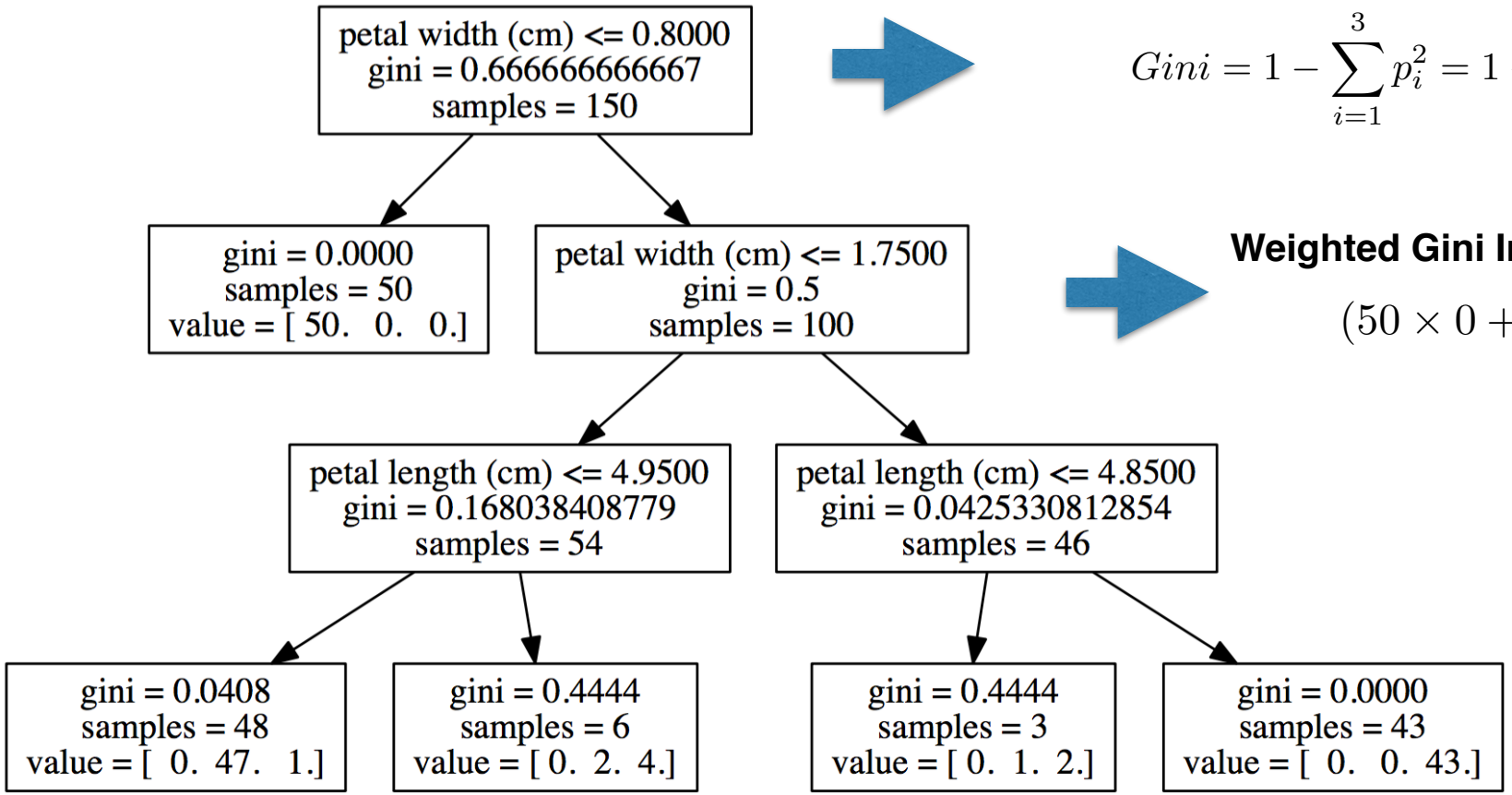
$$Gini = \sum_{i=1}^{m} P(c_i)(1 - P(c_i)) = 1 - \sum_{i=1}^{m} P(c_i)^2$$

Gini Impurity Example



With two classes, the impurity of a node is p(p-1), where p is the probability of the positive class

Iris Decision Tree



50 examples each of 3 classes.

$$Gini = 1 - \sum_{i=1}^{3} p_i^2 = 1 - (.33)^3 = .67$$

Weighted Gini Impurity across nodes

$$(50 \times 0 + 100 \times .5)/150$$

Choosing the best split

- For continuous variables, check every possible split point.
  - e.g. If a variables takes on the values: [2.1, 2.12, 3.4, 4.0, 5.1], you would try y <= (2.1, 2.12, 3.4, 40)
- For categorical variables, check every possible binary split.
  - In practice, calculate the impurity of each individual category level, then order them by impurity and check each possible split point of the sequence.

Code for preceding example

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris
from sklearn import tree
from sklearn.externals.six import StringIO
import pydot

# Build the tree
data = load_iris()
clf = DecisionTreeClassifier(max_depth=3)
X = data.data
y = data.target
clf.fit(X,y)

# Visualize
dot_data = StringIO()
tree.export_graphviz(clf, out_file=dot_data,
                     feature_names=data.feature_names)
graph = pydot.graph_from_dot_data(dot_data.getvalue())
graph.write_pdf('iris_tree.pdf')
```

TreeBasedMethods.ipynb
Classification Tree Visualization

Regression Trees

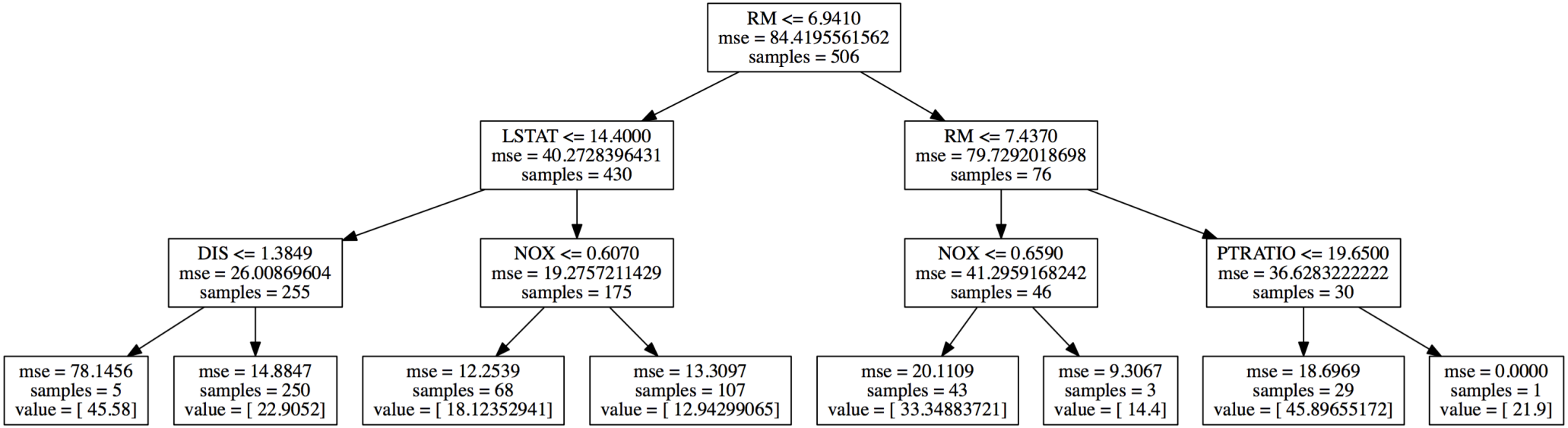
- In order to build a regression tree, one just needs a continuous splitting criterion
- Typically Mean-Squared-Error (MSE) is used
- Each node has a mean value - which becomes the predicted value.
- The error in an observation is the difference between the observed value and the mean value for that node (leaf)

Example: Boston Housing

Feature	Description
CRIM	per capita crime rate by town
ZN	proportion of residential land zoned for lots over 25,000 sq.ft.
INDUS	proportion of non-retail business acres per town
CHAS	Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
NOX	nitric oxides concentration (parts per 10 million)
RM	average number of rooms per dwelling
AGE	proportion of owner-occupied units built prior to 1940
DIS	weighted distances to five Boston employment centres
RAD	index of accessibility to radial highways
TAX	full-value property-tax rate per $10,000
PTRATIO	pupil-teacher ratio by town
B	Measure of racial diversity
LSTAT	% lower status of the population
MEDV	Median value of owner-occupied homes in $1000's

Goal is to predict median value of homes in a neighborhood

Boston Housing Example



IF
RM <= 6.94,
LSTAT > 14.4
NOX > 0.607
THEN
MEDV = 12.94 ($12,940)

Code for preceding slide - TreeBasedMethods.ipynb

Regression Example

In [2]:
```
from sklearn.tree import DecisionTreeRegressor
from sklearn.datasets import load_boston
from sklearn import tree
from sklearn.externals.six import StringIO
import pydot

# Build the tree
data = load_boston()
clf = DecisionTreeRegressor(max_depth=3)
X = data.data
y = data.target
clf.fit(X,y)

# Visualize
dot_data = StringIO()
tree.export_graphviz(clf, out_file=dot_data,
                     feature_names=data.feature_names)
graph = pydot.graph_from_dot_data(dot_data.getvalue())
graph.write_pdf('boston_tree.pdf')
```

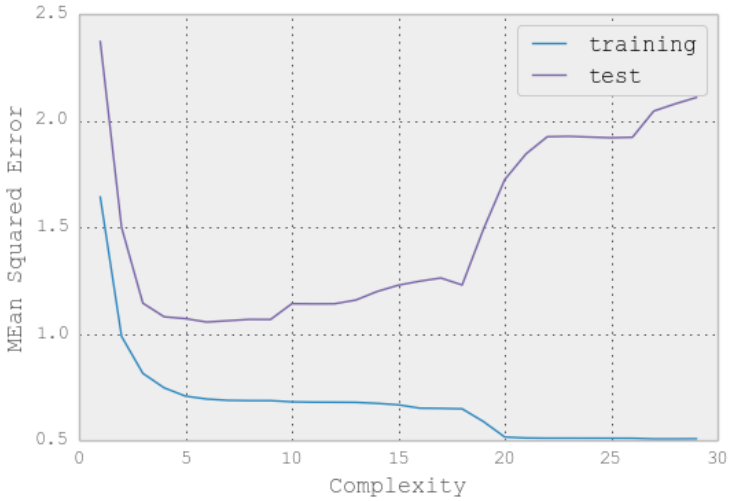
Out[2]: True

Overfitting

- We haven't talked about when you should stop recursively splitting the data.
- In theory, you could keep splitting until all of the leaves are 'pure', but this would lead to serious overfitting.
- In practice, we usually set a threshold on the *depth of the tree* or *number of samples per leaf* - this is called pre-pruning.

Post-Pruning

- The best approach called post-pruning.
- Instead of minimizing the impurity of the tree, minimize the impurity plus a complexity constant times the number of leaves in the tree.
- This has the effect forcing a different sub-tree for different values of the complexity parameter.



In practice, this approach is not even implemented in Scikit-Learn (although it is in R). The reason is that when prediction accuracy is at stake, other methods are usually better

Decision Tree Results - TreeBasedMethods.ipynb

Accuracy

Regression Tree

In [3]:
```
from sklearn.tree import DecisionTreeRegressor
from sklearn.datasets import load_boston
from sklearn.cross_validation import cross_val_score
from sklearn.cross_validation import train_test_split
from sklearn.grid_search import GridSearchCV

# Load Boston data
data = load_boston()

# Split into test/train
X_train, X_test, y_train, y_test = train_test_split(data.data, data.target,
                                                    test_size=.33,
                                                    random_state=0)

# Parameter Search
model = DecisionTreeRegressor()
depth_parm = np.linspace(1,12,12)
num_samples_parm = np.linspace(5,100,20)
parameters = {'max_depth' : depth_parm,
              'min_samples_leaf' : num_samples_parm}
regressor = GridSearchCV(model, parameters, scoring = 'mean_squared_error', cv=10)
regressor.fit(X_train,y_train)


# Test Prediction
pred = regressor.predict(X_test)
rmse = np.sqrt(np.mean((y_test - pred)**2))
rmse
```

Out[3]: 4.7253159917435754

Optimal parameters for max_depth and min_samples_leaf result in RMSE of $4,725

Decision Tree Tradeoffs

Why Decision Trees

- Easily Interpretable
- Handles missing values and outliers
- Non-linear / model complex phenomena
- Computationally cheap to *predict*
- Can handle irrelevant features
- Can handle mixed data (nominal and continuous)

Why Not Decision Trees

- Computationally expensive to *train*
- Greedy algorithm (can get stuck on local optima)
- Very easy to overfit

Break!

Ensemble Methods

Ensemble: A technique for combining many *weak* learners in an attempt to produce a *strong* learner.

Basic Idea: Build multiple models on the data and aggregate the predictive results to produce an overall prediction that is better than any of the individual models could do on their own.

Intuition

Classifier Example with Majority Vote:

- Suppose we have 5 completely independent classifiers with an accuracy of 70% for each. Then we can calculate the probability that they vote correctly:
  - $\binom{5}{3}(0.7)^3(0.3)^2 + \binom{5}{4}(0.7)^4(0.3)^1 + \binom{5}{5}(0.7)^5(0.3)^0 = 0.87$
  - yields 83.7% chance that majority vote is accurate
- 101 such classifiers would yield 99.9% majority vote accuracy

Ensemble learning works well to reduce variance, but it cannot reduce bias

Bagging

- Start with a dataset of size *n.*
- Sample from your dataset with replacement to create one bootstrap sample of size *n*, which means many of the observations will be repeated
- Repeat *B* times.
- Each bootstrap sample can then be used as a separate dataset for estimation or model fitting
- Simplest way to build your population of *weak learners.*

Random Forest

- Bagging improves prediction accuracy, but it can create problems when all of the trees are very similar.
  - The majority vote of your ensemble is not worth much if each individual has the same bias.
- **Solution:** De-correlate the trees by only selecting from a random subset of features at each ***split***.
  - Usually $\sqrt{n}$ features are chosen from at each split (out of n total features).
  - This is a tunable parameter

Random Forest: Parameters

The most important parameters to consider when building a random forest are *number of individual trees* and *number of features to choose at each split.*

num_trees: The higher the better, although computational time will be longer. Random forests are fairly resistant to overfitting, but gains will level out as this parameter increases.

max_features: A larger number tends to work well for regression (as opposed to classification). In general, decreasing max_features increases bias of the individual trees, but the resulting de-correlation accounts for this.

In theory, one can also adjust individual decision tree parameters, but one wants to minimize bias and maximize variance—often 1-2 samples per leaf works best

Random Forest - TreeBasedMethods.ipynb

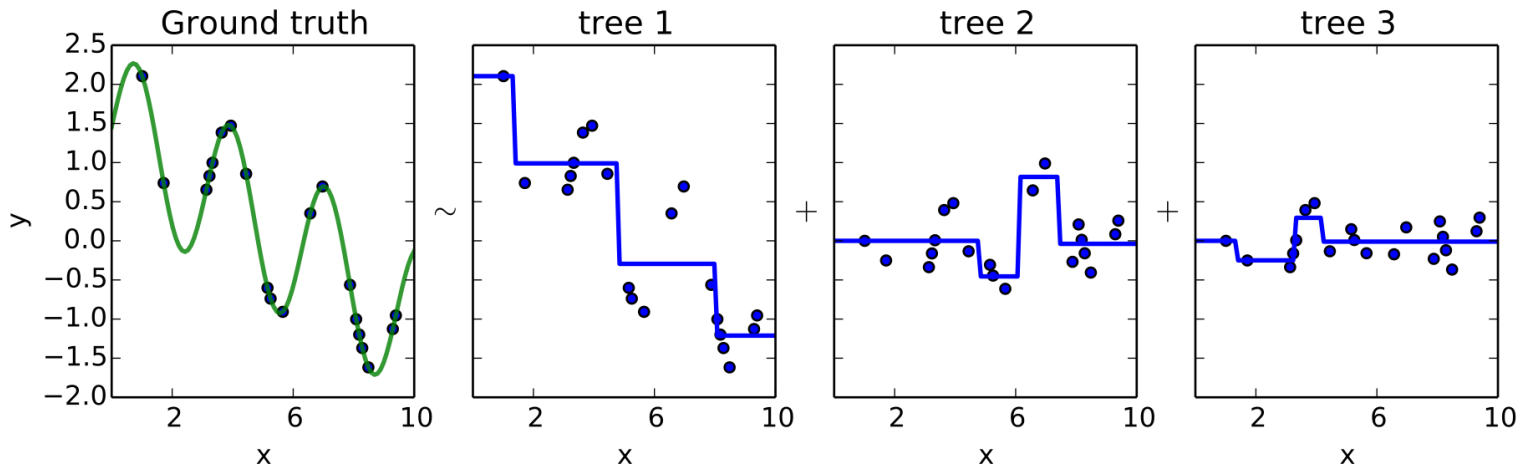
Random Forest

In [4]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.datasets import load_boston
from sklearn.cross_validation import train_test_split

# Load Boston data
data = load_boston()

# Split into test/train
X_train, X_test, y_train, y_test = train_test_split(data.data,
                                                    data.target,
                                                    test_size=.33,
                                                    random_state=0)

# Train and fit model
regressor = RandomForestRegressor(n_estimators=1000,
                                  max_features='auto',
                                  random_state=0)
regressor.fit(X_train,y_train)


# Test Prediction
pred = regressor.predict(X_test)
rmse = np.sqrt(np.mean((y_test - pred)**2))
rmse
```

Out[4]:
```
3.7281090856881427
```

Gradient Boosted Regression Trees (GBRT)

- Boosting is another way of combining many weak learners into a strong learner.
- Instead of building each tree individually, successive trees are built *on the error of the model thus far*.



GBRT - Overview

1. Build a (shallow) regression tree to fit your data. This becomes the *current* model.
2. For *B* iterations:
    1. Build a new tree on your data with the target replaced by the error between the truth and your current model's prediction.
    2. Update the current model by adding a scaled version of the new tree.
3. Output the current model.

GBRT

Algorithm 8.2 *Boosting for Regression Trees*

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

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

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

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

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

   (c) Update the residuals,

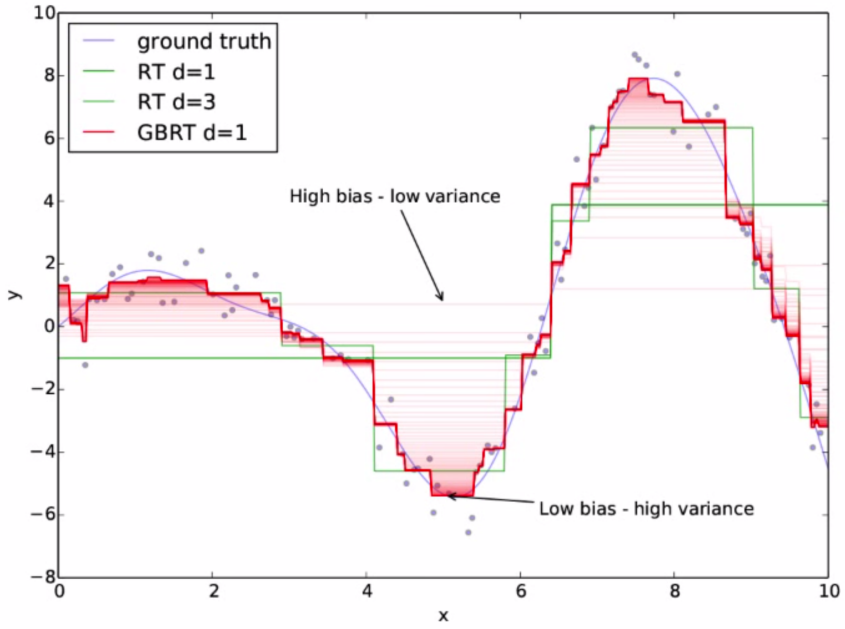
   $$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i). \tag{8.11}$$

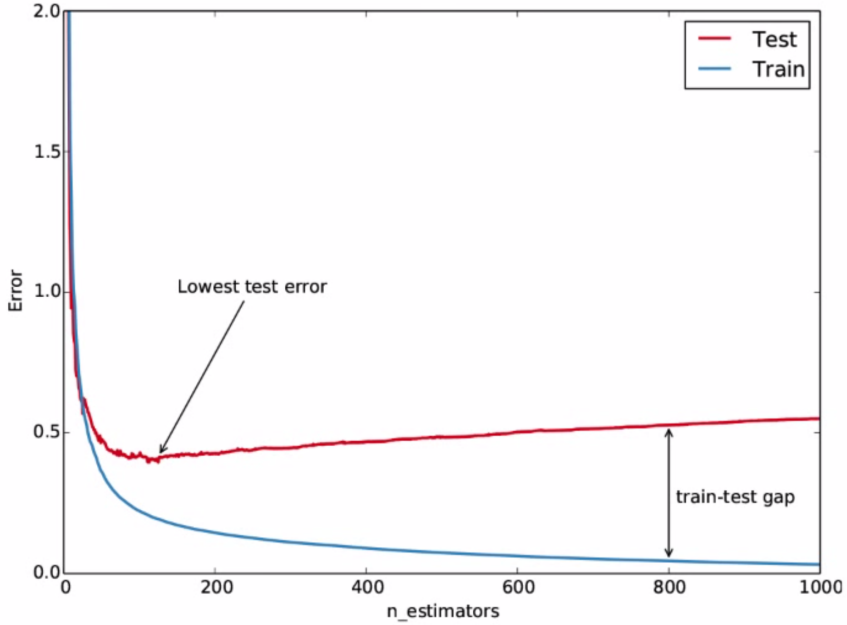
3. Output the boosted model,

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

From “An Introduction to Statistical Learning” - James, Witten, Hastie Tibshirani

GBRT





Boosting is not completely resistant to overfitting

GBRT - Parameters

- **Number of trees**: This should be quite high, but can result in overfitting
- **Learning Rate:** usually between .001 and .1. There is a trade-off between learning rate and number of trees.
- **Max Tree Depth:** Typically 1-6. This can effect bias/variance similar to with Random Forests
- **Stochastic?:** Stochastic GBRT occur when each tree is built with a bootstrap sample, and/or with subsets of features considered for splits.

GBRT - Results

Gradient Boosting

In [5]:
```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.datasets import load_boston
from sklearn.cross_validation import train_test_split

# Load Boston data
data = load_boston()

# Split into test/train
X_train, X_test, y_train, y_test = train_test_split(data.data,
                                                    data.target,
                                                    test_size=.33,
                                                    random_state=0)

# Parameter Search
gbr = GradientBoostingRegressor(n_estimators = 1000,
                                learning_rate=.1,
                                max_depth=4,
                                subsample=1,
                                max_features='auto',
                                random_state=0)
gbr.fit(X_train,y_train)


# Test Prediction
pred = gbr.predict(X_test)
rmse = np.sqrt(np.mean((y_test - pred)**2))
rmse
```

Only one line is different from random forest

Out[5]: 3.458634505688496

Comparison of Methods - On Boston Housing Data

	Root-Mean-Square-Error
Decision Tree	4.72
Random Forest	3.73
Gradient Boosted Regression Tree	3.46

Ensemble methods will always perform better than a single decision tree… but we have lost a great deal of *interpretability.*

Model Interpretation

- Model Interpretability is the most commonly cited drawback of using ensembles of trees.
- The standard approach to interpretation is to measure the importance of each variable. Two methods exist:
  1. Mean Decrease Impurity
  2. Mean Decrease Accuracy
- Another approach is to analyze decision paths for individual data points.

Mean Decrease Impurity

- For each tree, each split is made in order to reduce the total impurity of the tree (Gini Impurity for classification, RSS for regression); we can record the magnitude of the reduction.
- Then the importance of a feature is the average decrease in impurity across trees in the forest, as a result of splits defined by that feature.
- **GOAL:** To determine which features have the largest impact on the model.

MDI - scikit-learn

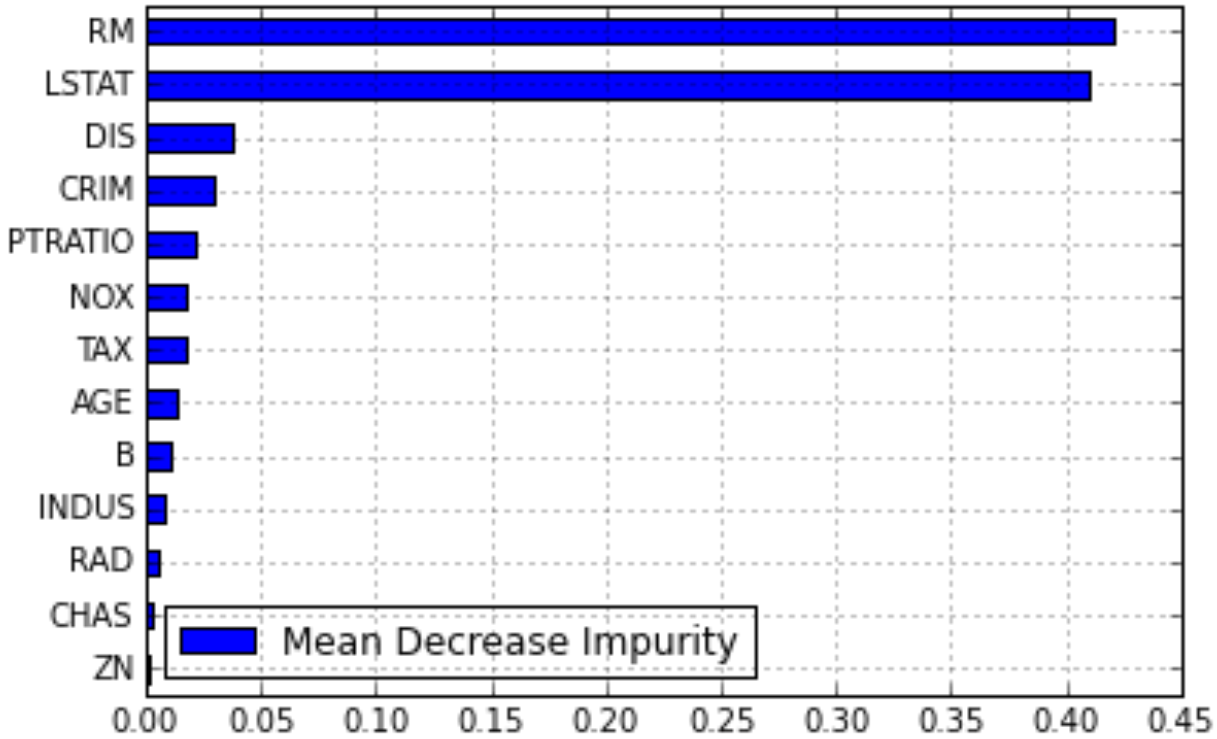
Random Forest Interpretation

Mean Decrease Impurity

In [9]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.datasets import load_boston
from collections import defaultdict
import numpy as np
import pandas as pd

data = load_boston()
rf = RandomForestRegressor(n_estimators=1000, random_state=0)

# Split into test_train
train, test, target, actual = train_test_split(
    data.data, data.target, test_size=.33, random_state=0)

# Fit the model
rf.fit(train, target)
prediction = rf.predict(test)

# Plot the feature importance
feat_scores = pd.DataFrame({'Mean Decrease Impurity' : rf.feature_importances_},
                           index=data.feature_names)
feat_scores = feat_scores.sort('Mean Decrease Impurity')
feat_scores.plot(kind='barh')
```

Out[9]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x109faacd0>
```

MDI - Boston Housing data



The biggest factors in predicting the value of a neighborhood are the average number of rooms and the class status of the neighborhood

Mean Decrease Accuracy

- MDI tends to be biased to continuous features and/or categorical features with many levels
- Another approach:
  - Calculate the accuracy of your model on a test set.
  - Randomly permute the values of feature X in your test set.
  - Recompute the accuracy and see how much it has decreased.

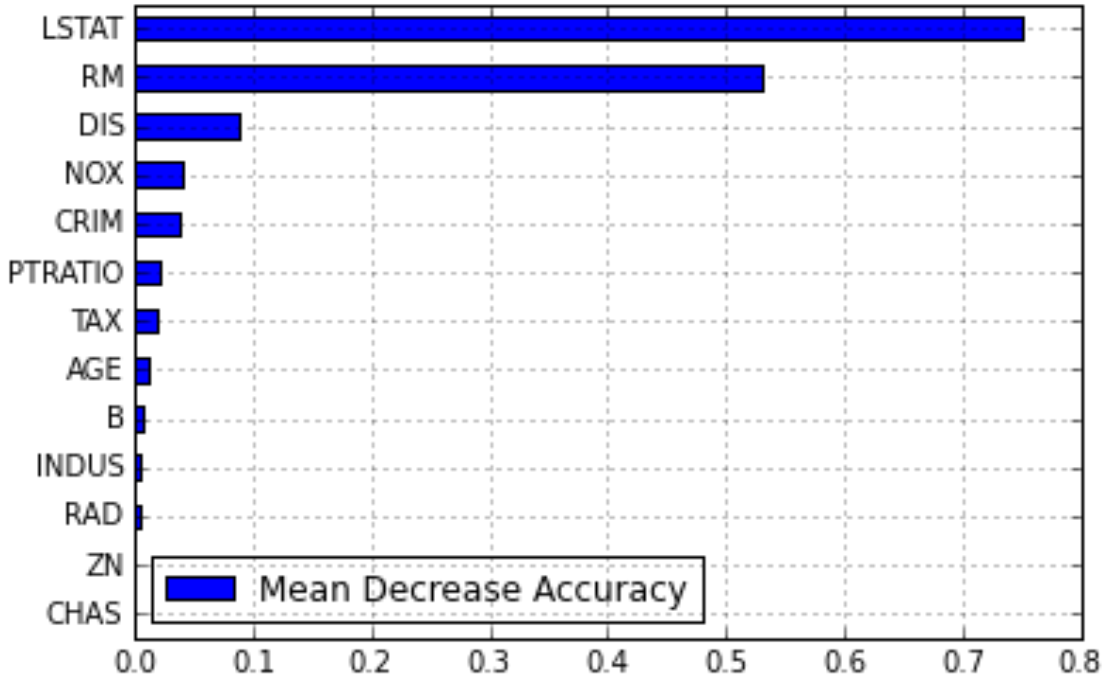
MDA - scikit-learn
See: http://blog.datadive.net/

Mean Decrease Accuracy

In [10]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.cross_validation import ShuffleSplit
from sklearn.metrics import r2_score
from collections import defaultdict

boston = load_boston()
names = boston.feature_names
X = boston["data"]
Y = boston["target"]

rf = RandomForestRegressor()
scores = defaultdict(list)

# crossvalidate the scores on a number of
# different random splits of the data
for train_idx, test_idx in ShuffleSplit(len(X), 100, .3):
    X_train, X_test = X[train_idx], X[test_idx]
    Y_train, Y_test = Y[train_idx], Y[test_idx]
    r = rf.fit(X_train, Y_train)
    acc = r2_score(Y_test, rf.predict(X_test))
    for i in range(X.shape[1]):
        X_t = X_test.copy()
        np.random.shuffle(X_t[:, i])
        shuff_acc = r2_score(Y_test, rf.predict(X_t))
        scores[names[i]].append((acc-shuff_acc)/acc)

score_series = pd.DataFrame(scores).mean()
scores = pd.DataFrame({'Mean Decrease Accuracy' : score_series})
scores.sort('Mean Decrease Accuracy').plot(kind='barh')
```

Out[10]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x109f62cd0>
```

MDA - Boston Housing Data

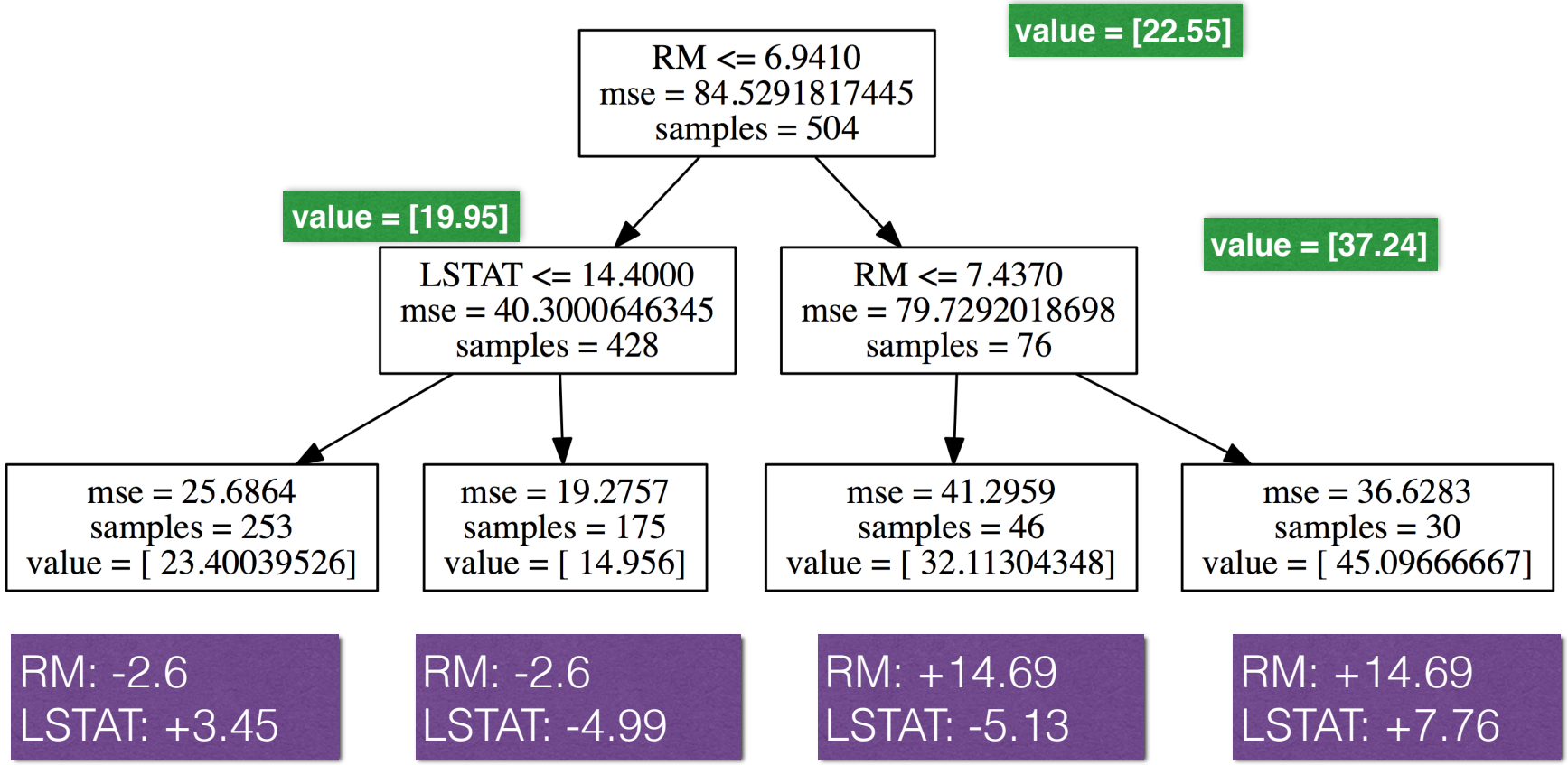


Results are similar to MDI, but some of the relative magnitudes are different

Decision Tree Paths

- There is another choice for interpreting Random Forests (or other tree ensembles)
- When a tree makes a prediction about a new observation, we can think of that observation following a path down the tree.
- Keeping track of these paths gives additional information about how the prediction is actually made.

Path Contributions Example



Any prediction can be decomposed into its gain/loss because of RM + its gain/loss due to LSTAT

Decision Paths Example

In [3]:
```
from sklearn.tree import DecisionTreeRegressor
from sklearn.datasets import load_boston
from collections import defaultdict
import numpy as np



data = load_boston()
dt = DecisionTreeRegressor(max_depth=4)
# dt = RandomForestRegressor()

#Compute fature contributins on three samples. Use the rest of the data
#as training set
train = data.data[:-3]
target = data.target[:-3]
test = data.data[-3:]
actual = data.target[-3:]

dt.fit(train, target)
prediction = dt.predict(test)

#compute the decision paths from root to leaf for each sample
paths = dt.decision_paths(test)
contribution_list = []
for path in paths:
    contributions = defaultdict(int)
    for i in range(len(path)):
        if i == len(path) - 1 or path[i+1] == -1:
            break
        node_id = path[i]
        next_node_id = path[i+1]

        #feature contribution at a node is the difference of the mean of the
        #decision node, and the mean at the following child node
        contributions[dt.tree_.feature[node_id]] += \
            dt.tree_.value[next_node_id][0][0] - dt.tree_.value[node_id][0][0]
    contribution_list.append(contributions)

print "Training set mean:", np.mean(target)
print
for i, contribs in enumerate(contribution_list):
    print "Prediction %s: %s " %(i + 1, prediction[i])
    print "Actual %s: %s " %(i+1, actual[i])
    print "Top five contributing features:"
    clist = sorted(contribs.items(), key = lambda x: -abs(x[1]))[:5]
    for key, val in clist:
        print data.feature_names[key], val
    print
    if i > 1:
        break
```

Output from one decision tree

Training set mean: 22.5522862823

Prediction 1: 32.97
Actual 1: 23.9
Top five contributing features:
RM 9.74326927325
NOX 1.27825396825
DIS -0.60380952381

Prediction 2: 27.5277777778
Actual 2: 22.0
Top five contributing features:
LSTAT 3.45273170552
RM 1.96992924042
DIS -0.447169450465

Prediction 3: 21.6798969072
Actual 3: 11.9
Top five contributing features:
RM -3.87795163014
LSTAT 3.45273170552
DIS -0.447169450465

Random Forest Decision Paths

- Decision paths can be naturally extended to Random Forests
- Whenever a prediction is made, the contribution paths for each tree are averaged across the forest.

Random Forest Decision Path Example

1. Split Boston data into train/test and build a Random Forest Regression model on the training data.
2. Pick 6 observations in the test set that have similar predictions (all ~ $30K).
3. Calculate the average contribution, from the model, across all trees, for each observation.

Raw Data

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
Observation 1	9.23230	0	18.10	0	0.631	6.216	100.0	1.1691	24	666	20.2	366.15	9.53
Observation 2	8.26725	0	18.10	1	0.668	5.875	89.6	1.1296	24	666	20.2	347.88	8.88
Observation 3	0.03049	55	3.78	0	0.484	6.874	28.1	6.4654	5	370	17.6	387.97	4.61
Observation 4	0.04417	70	2.24	0	0.400	6.871	47.4	7.8278	5	358	14.8	390.86	6.07
Observation 5	0.06911	45	3.44	0	0.437	6.739	30.8	6.4798	5	398	15.2	389.71	4.69
Observation 6	0.49298	0	9.90	0	0.544	6.635	82.5	3.3175	4	304	18.4	396.90	4.54

Looking at the raw data can tell you the difference between the different observations, but it doesn't tell you *how* and *why* they are all predicted to have the same value.

Average of contribution paths

Average thousands of dollars added to the estimate, by the model, because of the value of AGE.

Distance from city-center

Fraction of lower-class residents

Average number of rooms

	AGE	B	CHAS	CRIM	DIS	INDUS	LSTAT	NOX	PTRATIO	RAD	RM	TAX	ZN
Prediction 1	-0.09	0.12	-0.02	0.53	6.11	0.10	2.67	-0.02	-0.19	0.06	-2.63	-0.20	0.03
Prediction 2	-0.07	0.04	0.05	0.63	6.22	0.16	2.56	-0.01	-0.19	0.06	-3.15	-0.16	0.03
Prediction 3	-0.01	-0.12	-0.05	0.05	-0.70	0.04	7.42	-0.11	0.42	-0.02	1.10	-0.14	-0.05
Prediction 4	0.04	-0.14	-0.02	-0.02	-0.53	0.25	3.50	0.16	1.46	0.13	2.21	-0.24	0.11
Prediction 5	0.17	-0.13	-0.02	-0.03	-0.68	0.15	7.86	0.03	0.85	0.01	-1.14	-0.16	0.18
Prediction 6	0.60	-0.05	-0.03	0.03	0.18	-0.26	8.62	-0.19	-0.02	-0.07	-1.83	-0.34	-0.05

All 6 houses are predicted to have a value of $30K

- Predictions 1 and 2 are based primarily on those houses being close to the city center
- Prediction 4 is a relatively large house in a relatively good neighborhood.
- Predictions 5 and 6 are smaller houses, but in really nice neighborhoods.

Clustering Decision Paths

1. Pick a handful (I chose 3) of the most important variables—-chosen by Mean Decrease Impurity.
2. Compute average decision paths for all the data.
3. Cluster (e.g. using k-means) on the average contributions of the top variables.
4. Examine the clusters for insight into both the model and your data!

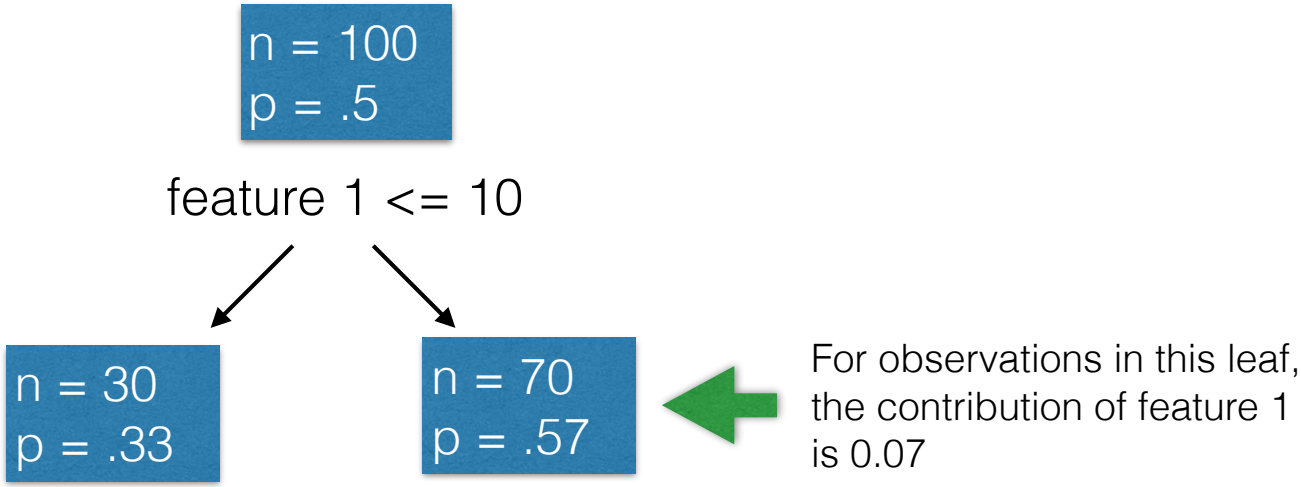
Clusters

	DIS	LSTAT	RM	Mean Predicted Value	Count	Description
cluster_id						
0	0.24	-5.85	-1.50	16.85	106	Small houses in bad neighborhoods
1	0.11	2.05	-2.84	22.33	58	Small houses in better neighborhoods
2	0.10	5.50	16.23	45.15	29	Huge houses in nice neighborhoods
3	-0.27	2.92	7.56	33.52	29	Large houses in decent neighborhoods
4	-0.73	-7.07	-1.53	11.28	65	Small houses in terrible neighborhoods
5	-0.01	-0.10	-2.36	20.10	100	Small houses in average neighborhoods
6	0.15	3.16	1.11	27.50	39	Average houses in decent neighborhoods
7	8.75	8.66	-0.14	45.89	3	Normal size houses, in great neighborhood close to city center
8	-0.25	6.07	-4.18	24.35	55	Small houses in very nice neighborhoods
9	-0.42	7.59	1.05	31.84	22	Normal size houses in very nice neighborhoods

Even though RF is a ‘black-box’ we know, roughly how it is making predictions about each observation.

Decision Paths for Classification Problems

- For binary classification, the contribution of a feature responsible for a split is the change in proportion of the positive class
- Multi-class classification decision paths are vectorized—there is one contribution for each outcome class.



Conclusions

- Decision trees are highly flexible modeling tools, that can be applied to virtually any prediction problem. (and with very little pre-processing of the data)
- Ensembles of decision trees increase accuracy dramatically (on-par with SVMs and Neural Networks), but one loses some interpretability.
- With a little bit of effort tree-ensembles can actually be interpreted quite effectively.
- Check us out! www.galvanize.com