STAT 177, CLASS 8

Richard Waterman

July 2020

OBJECTIVES

OBJECTIVES

- Machine learning introduction.
- Decision trees.
  - Regression.
  - Classification
- Assessing the accuracy of a classifier.
- Ensemble learners – the random forest.

MACHINE LEARNING

MACHINE LEARNING

- Statistical modeling and machine learning have huge overlapping areas of interest.
- Oftentimes machine learning focuses on feature engineering and *prediction* .
- Statistical modeling, has a focus on interpretation and the appropriate measurement of uncertainty (confidence intervals and p-values).
- Problems are split into two types:
  - Supervised learning is when there is an outcome (y-variable). Methods here include regression and tree models.
  - Unsupervised learning is when there's no outcome variable. Methods include clustering and dimension reduction.
- We will focus on supervised methods.

SOME NOTATION

- The response variable, the one we are trying to predict is usually denoted as the Y -variable.
- The predictor variables (features) are usually written as $X_1, X_2 \cdots X_k$.
- The general idea is to use the features in some optimal way to predict the Y -variable.
- When the response is a continuous variable, a popular measure of “optimal” is to minimize $SSE = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$, where $\hat{y}_i$ is the predicted value of the i-th observation and SSE stands for the “Sum of the Squares of the Errors”.
- Sometimes this criterion os called *squared error loss* or the L_2 loss function.
- The SSE/n , where n is the sample size is called the “Mean squared error” (MSE).
- $\sqrt{MSE}$ is called, not surprisingly, Root Mean Squared Error (RMSE).

CATEGORICAL OUTCOMES

- When the outcome is a dichotomous variable (0/1) a commonly used loss function is “logistic”, sometimes called “cross-entropy loss”:

$$-\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)\right].$$

Here y_i and $\hat{y}_i$ are the observed, and predicted values of the i-th observation respectively.

DECISION TREES

DECISION TREES

- These are a foundational methodology that other techniques are also built from.
- There's no model, and the tree is built by following an algorithm that recursively partitions the feature (X) space, so that the outcome is as homogeneous as possible within the partitions.
- To get some intuition we will start with a continuous y-variable and see how a single split is made.
- If we can understand a single split, then because it is a "recursive" algorithm, we can understand all the splits!
- We will dig into trees a bit more in the Stat notes.

SKLEARN IMPLEMENTATION OF DECISION TREES

SKLEARN IMPLEMENTATION OF DECISION TREES

- We will use the functionality available in the the sklearn machine learning library to build a decision tree.
- We will start with the same data that was used in the “Decision tree” notes, with the true mean following a sine curve, then open up some real datasets.

```
import os
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import tree
import numpy as np

os.chdir('C:\\Users\\richardw\\Dropbox (Penn)\\Teaching\\477s2020\\DataSets')
example_data = pd.read_csv("treealgo.csv")
```

REVIEW THE DATA FRAME

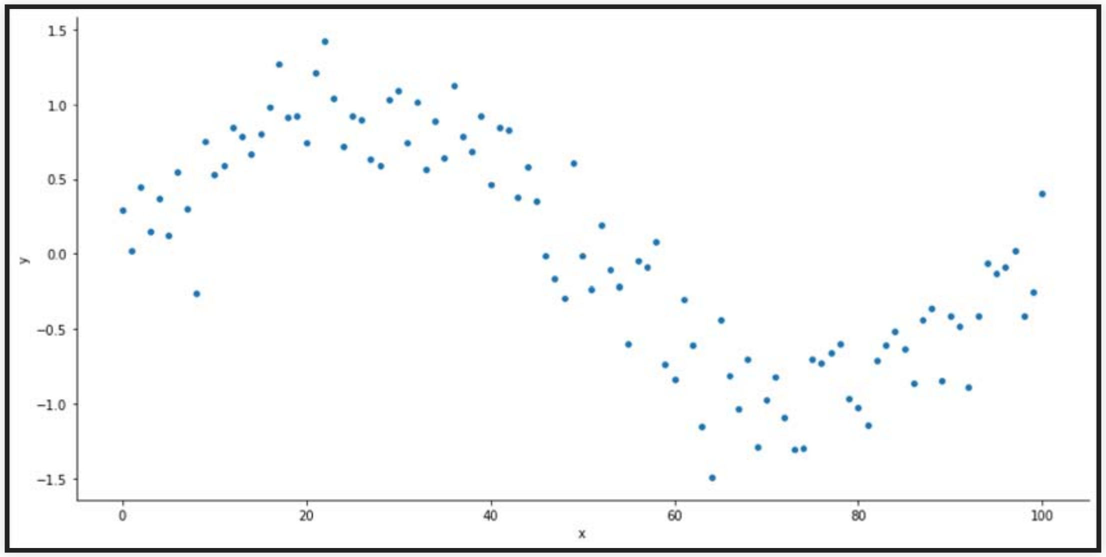
```
print(example_data.info()) # Summarizing the dataset with the ".info()" method
print(example_data.head(3))
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 101 entries, 0 to 100
Data columns (total 3 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Unnamed: 0  101 non-null    int64
 1   y           101 non-null    float64
 2   x           101 non-null    int64
dtypes: float64(1), int64(2)
memory usage: 2.5 KB
None
   Unnamed: 0         y  x
0           1  0.296105  0
1           2  0.019676  1
2           3  0.441736  2
```

PLOTTING THE RAW DATA

- Here's a plot of the raw data for which we want to build a predictive model.

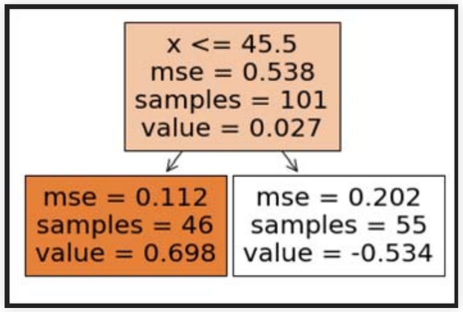
```
sns.relplot(x = 'x', y = 'y', data=example_data,height=6,aspect=2);
```



FITTING THE TREE

- The tree is a collection of “if/else” rules.
- In the terminal nodes the *MSE* , number of observations and the predicted value for that node are reported.

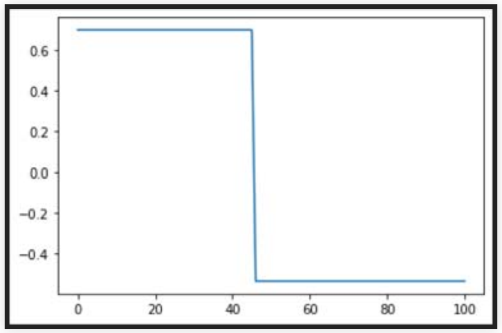
```
X = example_data[['x']] # Two brackets [[ ]] is important here because the X-variable needs to be
  matrix like.
y = example_data['y']
regtree = tree.DecisionTreeRegressor(max_leaf_nodes = 2) # Make a single split to match the D"ecision
  Tree" notes.
regtree = regtree.fit(X, y)
tree.plot_tree(regtree, filled=True, feature_names=X.columns);
```



PREDICTION

- Prediction happens with the .predict() method.
- It expects the X variable(s) to be in a 2D array.
- The ‘reshape’ method is turning the 1D array into a 2D matrix, albeit with one column.
- But the seaborn line-plot expects a one-dimensional array (vector) as input.

```
preds = regtree.predict(np.arange(0,101).reshape(-1, 1))
sns.lineplot(x = np.arange(0,101), y = preds);
```



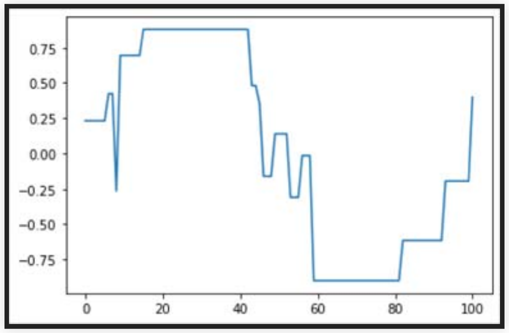
INCREASING THE NUMBER OF LEAVES

- Complexity of the tree can be managed in a number of ways including by setting:
  - max_depth: the maximum depth of the tree.
  - min_samples_split: the minimum number of samples required to split an internal node.
  - max_leaf_nodes: the maximum number of leaf nodes.
  - min_impurity_decrease: the impurity decrease must be greater than this value to continue splitting.
- We will fit a more complex tree and visualize the predictions.

INCREASING THE NUMBER OF LEAVES

- In the graphic below which plots the value of x against the prediction from the tree, you can see the underlying sine curve starting to be revealed as the complexity of the tree increases.

```
regtree = tree.DecisionTreeRegressor(max_depth = 4) # Allow a depth of at most 4.
regtree = regtree.fit(X, y)
preds = regtree.predict(np.arange(0,101).reshape(-1, 1))
sns.lineplot(x = np.arange(0,101), y = preds);
```



PREDICTING FUEL ECONOMY

- The car data set has a number of categorical variables.
- The tree fitting algorithm expects a fully numerical X-matrix as input.
- Therefore we have to encode the categorical variables numerically.
- We will use dummy variable encoding, available via pandas “get_dummies” function.
- In computer science speak, the dummy variables are often called a “one-hot encoding”.

```
carTable = pd.read_csv("Car08_just_499.csv")

# Pull off the continuos variables first.
Xcts = carTable[['Weight(lb)', 'Seating', 'Horsepower', 'HP/Pound', 'Displacement', 'Cylinders',
  'Length']]
# Deal with the categorical predictors by making then into dummy variables using "get_dummies".
Xcat = pd.get_dummies(carTable[['Transmission', 'EPA_Class','HEV', 'Turbocharger']],drop_first=True) #
  Categorical variables.

# Merge the continuous and categorical variables to a single dataset.
X = pd.merge(Xcts, Xcat, left_index=True, right_index=True)
y = carTable['GP1000M_City']
```

REVIEW THE NEW DATA FRAME

- Note the dummy variable encoding visible in the data frame.

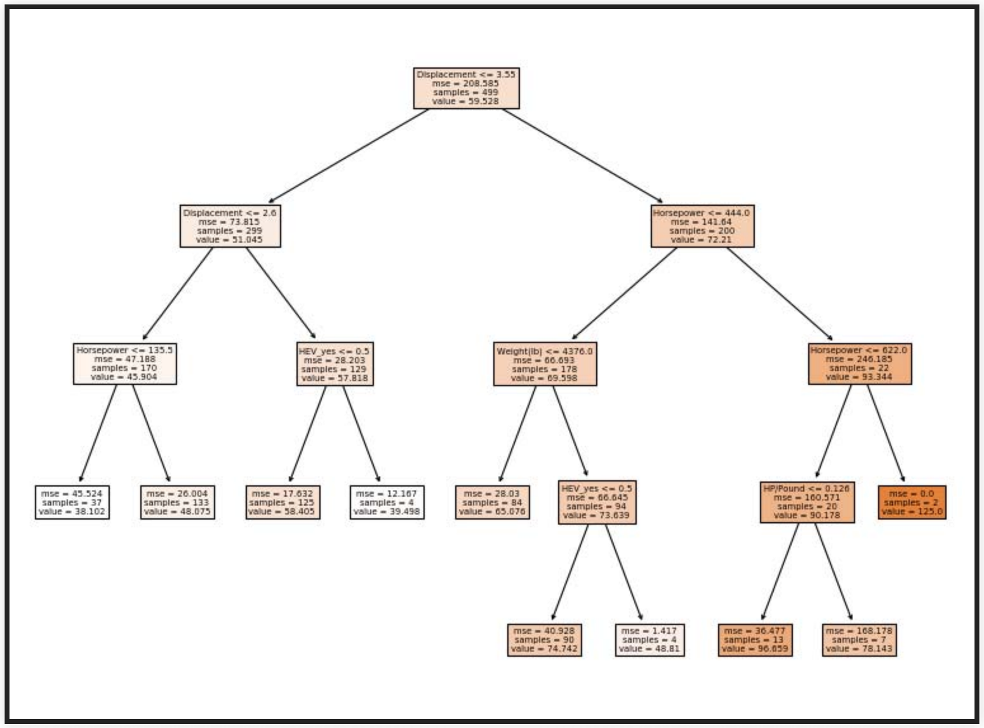
```
print(X.head(5))
```

```
   Weight(lb)  Seating  Horsepower  HP/Pound  Displacement  Cylinders  Length  \
0        4014        5         290  0.072247           3.5          6   193.6
1        3674        5         286  0.077844           3.5          6   189.3
2        3559        5         286  0.080360           3.5          6   189.3
3        3345        5         205  0.061285           2.4          4   186.2
4        3257        5         205  0.062941           2.4          4   186.2

   Transmission_AS  Transmission_AV  Transmission_M  ...  EPA_Class_midsize  \
0                1                0               0  ...                  1
1                1                0               0  ...                  1
2                0                0               1  ...                  1
3                1                0               0  ...                  0
4                0                0               1  ...                  0

   EPA_Class_midsizewagon  EPA_Class_minicompact  EPA_Class_minivan  \
```

FITTING A TREE

- Below we fit a tree, controlling complexity by the maximum number of leaf nodes.
- The “.plot_tree()” has many more arguments that you can tweak.

```
regtree = tree.DecisionTreeRegressor(max_leaf_nodes = 10) # Set up the tree structure.
regtree = regtree.fit(X, y) # Fit the tree.

fig = plt.figure(num=None, figsize=(12, 9), dpi=80, facecolor='w', edgecolor='k')
tree.plot_tree(regtree, filled=True, feature_names=X.columns);
```


Displacement <= 3.55
mse = 208.585
samples = 499
value = 59.528
Displacement <= 2.6
mse = 73.815
samples = 299
value = 51.045
Horsepower <= 444.0
mse = 141.64
samples = 200
value = 72.21
Horsepower <= 135.5
mse = 47.188
samples = 170
value = 45.904
HEV_yes <= 0.5
mse = 28.203
samples = 129
value = 57.818
Weight(lb) <= 4376.0
mse = 66.693
samples = 178
value = 69.598
Horsepower <= 622.0
mse = 246.185
samples = 22
value = 93.344
mse = 45.524
samples = 37
value = 38.102
mse = 26.004
samples = 133
value = 48.075
mse = 17.632
samples = 125
value = 58.405
mse = 12.167
samples = 4
value = 39.498
mse = 28.03
samples = 84
value = 65.076
HEV_yes <= 0.5
mse = 66.645
samples = 94
value = 73.639
HP/Pound <= 0.126
mse = 160.571
samples = 20
value = 90.178
mse = 0.0
samples = 2
value = 125.0
mse = 40.928
samples = 90
value = 74.742
mse = 1.417
samples = 4
value = 48.81
mse = 36.477
samples = 13
value = 96.659
mse = 168.178
samples = 7
value = 78.143

CHOOSING THE BEST TREE

- Most modeling involves a trade-off between making things too simple (underfit and biased) and too complicated (overfit and high variance).
- This is known as the “bias/variance” trade-off.
- The problem with overfitting is that the predictions do not generalize well to new data.
- The problem with underfitting is that the predictions are biased.
- When growing a tree it is possible to trade these concerns off against one another, by using “cost-complexity” pruning.
- This type of pruning effectively removes the “weakest link” in the tree, recursively, providing a set of trees to explore for the best generalizability.
- The idea is to grow an overly complicated tree (to make sure we don’t miss any important splits), then to “prune” it back so that it generalizes as well as possible.
- We measure a tree’s generalizability through “cross-validation” or how well it actually predicts when it sees new data.

THE TRAIN AND TEST PARADIGM

THE TRAIN AND TEST PARADIGM

- This is a critical idea and involves splitting the data into two distinct parts.
- The portion that the model is fit on is called the *training data* .
- The portion that the model predictions are evaluated on, is called the *test data* .
- There is no magic rule for how much data to put into the training/test datasets, but anywhere between 80/20 to 50/50 is often used.
- The key idea for the test dataset, is that because it was not used to build the model, it provides a legitimate way of assessing the model’s accuracy.
- If you assessed accuracy on the training dataset you would be over-optimistic, because the model is fit to minimize the prediction error in the first place.

BREAKING THE DATA INTO TWO

- The key command here from sklearn is “train_test_split”.
- Always make sure you use a random number seed (random_state) for reproducibility.

```
# Break the data in 2 parts randomly
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.5, random_state=1) # A  50/50
  split.
# Confirm that this has worked.
print(X_train.shape)
print(X_test.shape)
```

```
(249, 22)
(250, 22)
```

COST COMPLEXITY PRUNING

- Denote a fitted tree as T and for a regression tree, the quality of the tree as R(T), where R is the sums of squared error (low values of R(T) mean a high quality tree). We want R(T) to be low, and it will get smaller as the tree becomes more complicated.
- Define $|T|$ as the number of leaves in the tree (terminal nodes). This is a way of defining the complexity of the tree.
- Let α be a complexity parameter that measures the ``cost'' of adding another node to the tree.
- Now define the “cost” of a tree as:

$$R_\alpha(T) = R(T) + \alpha|T|.$$

COST COMPLEXITY PRUNING (CTD.)

- This formulation provides a way of penalizing complexity. If α is very big, you are reticent to add nodes. If α is very small, adding nodes has a minimal price.
- For each value of α , there is a tree that minimizes cost. Call it T_α .
- Then for each tree, T_α , find its test set prediction error.
- Choose the α that has lowest test set prediction error.

IMPLEMENTATION

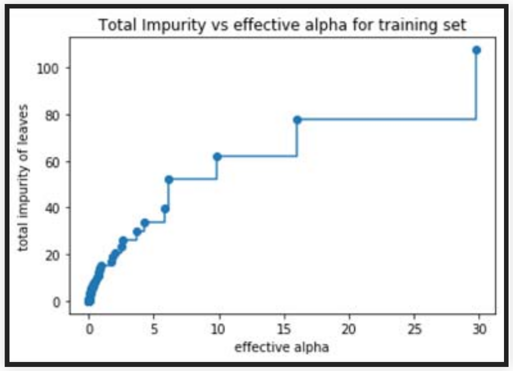
- To get the set of trees we will explore, use the “.cost_complexity_pruning_path()” function.
- Th eplot below shows how the nodes’ impurity increases with α .
- Simple trees have a lot of impurity.

```
dtr = tree.DecisionTreeRegressor(random_state=0) # Set up the tree.
path = dtr.cost_complexity_pruning_path(X_train, y_train) # Create the pruning path.
ccp_alphas, impurities = path.ccp_alphas, path.impurities # The tree impurities along the pruning path.
```

IMPLEMENTATION

```
fig, ax = plt.subplots()
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')
```



FITTING THE DECISION TREE FOR EACH VALUE OF α

- Recall that for each value of α there is a best decision tree.
- We now fit all those trees in a loop and store each tree in the list “rgrs”.
- We then see how well each tree predicts on the test data, and choose the tree with the best performance.

```
rgrs = [] # A container for the trees.
for ccp_alpha in ccp_alphas: # A for loop, fitting a tree for each value of alpha
    dtr = tree.DecisionTreeRegressor(random_state=0, ccp_alpha=ccp_alpha)
    dtr.fit(X_train, y_train)
    rgrs.append(dtr)
print("Number of nodes in the last tree is: {} with ccp_alpha: {}".format(
      rgrs[-1].tree_.node_count, ccp_alphas[-1]))
```

```
Number of nodes in the last tree is: 1 with ccp_alpha: 107.31669971836456
```

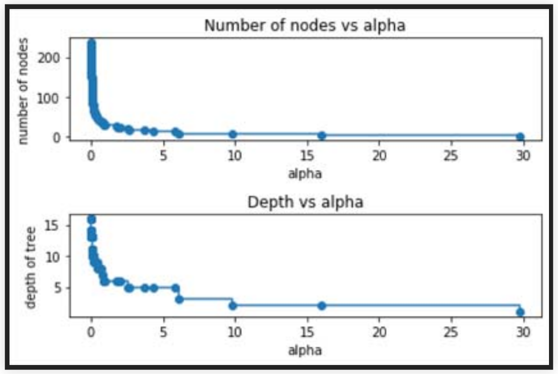
THE COMPLEXITY VERSUS ALPHA PLOTS

- Below is the plot of α (penalty parameter) against complexity (number of nodes).
- And, α (penalty parameter) against complexity (tree depth).
- The key aspect to note is that as alpha gets larger the tree is less complex.

```
rgrs = rgrs[:-1] # Remove the simplest tree (it has no branches).
ccp_alphas = ccp_alphas[:-1]

node_counts = [dtr.tree_.node_count for dtr in rgrs] # Note the list comprehensions.
depth = [dtr.tree_.max_depth for dtr in rgrs] # Note the list comprehensions
```

THE COMPLEXITY VERSUS ALPHA PLOTS

```
fig, ax = plt.subplots(2, 1)
ax[0].plot(ccp_alphas, node_counts, marker='o', drawstyle="steps-post")
ax[0].set_xlabel("alpha")
ax[0].set_ylabel("number of nodes")
ax[0].set_title("Number of nodes vs alpha")
ax[1].plot(ccp_alphas, depth, marker='o', drawstyle="steps-post")
ax[1].set_xlabel("alpha")
ax[1].set_ylabel("depth of tree")
ax[1].set_title("Depth vs alpha")
fig.tight_layout()
```



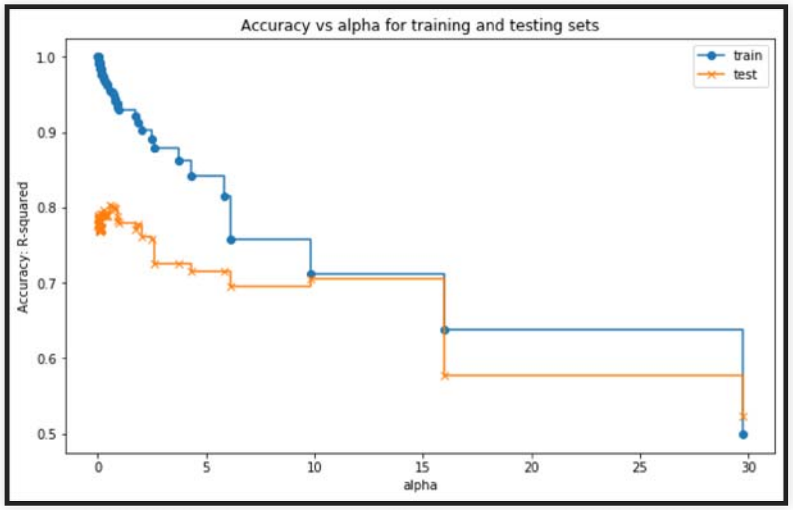
TRAINING V. TEST ERRORS ALONG THE PRUNING PATH

- As the outcome variable is continuous, the default choice for “Accuracy” in the below plot, is R^2 .
- It is obtained by “scoring” the tree.

```
# Obtain the "score" for each tree.

train_scores = [dtr.score(X_train, y_train) for dtr in rgrs] # The score function returns the R-squared
  here.
test_scores = [dtr.score(X_test, y_test) for dtr in rgrs] # The score function returns the R-squared
  here.
```

PLOTTING THE R^2 AGAINST ALPHA

```
fig, ax = plt.subplots(figsize=(10, 6))
ax.set_xlabel("alpha")
ax.set_ylabel("Accuracy: R-squared")
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='x', label="test",
        drawstyle="steps-post")
ax.legend()
plt.show()
```



LOOKING AT THE SCORES

- We are tying to find the top of the orange curve.
- That’s the value of α at which the test R^2 is maximized.
- Visually it happens at a low value for α on the left-hand-side of the plot.

FINDING THE BEST VALUE OF α

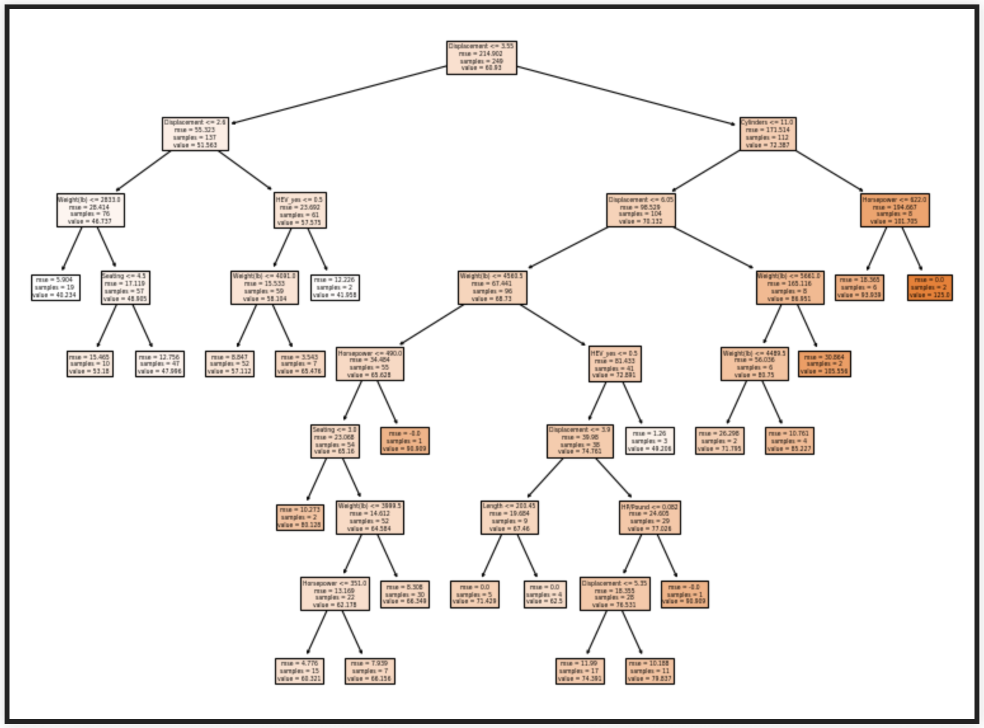
- Plan:
  - Find the tree with the best test score.
  - Pull it out of the tree list, and review it.

```
best = pd.Series(test_scores).idxmax() # Find the index of the best tree.
ccp_alphas[best] # here's the best tree's value of "alpha".
```

```
0.5720346935927063
```

REVIEWING THE BEST TREE

```
best_tree = rgrs[best] # Pull out the best tree from the pruning path.

fig = plt.figure(num=None, figsize=(12, 9), dpi=80, facecolor='w', edgecolor='k')
tree.plot_tree(best_tree, filled=True, feature_names=X.columns);
```



VARIABLE IMPORTANCE

- The decision tree comes with a variable importance metric.
- This helps you get a sense of which variables are most helpful in prediction.
- This can be useful if you are later going to build a regression model.
- The method is “.feature_importances_”
- Formally, the importance of a feature is computed as the (normalized) total reduction of the criterion brought by that feature.

OBTAINING THE VARIABLE IMPORTANCE RANKING

```
importances =
  pd.DataFrame({'feature':X_train.columns,'importance':np.round(best_tree.feature_importances_,3)})
importances = importances.sort_values('importance',ascending=False)
print(importances[:5]) # Top five variables.
```

```
         feature  importance
4   Displacement       0.664
5      Cylinders       0.145
0     Weight(lb)       0.081
20       HEV_yes       0.045
2     Horsepower       0.044
```

REVIEW

- We now have an automatic means of creating a predictive model, using a decision tree and cost/complexity pruning.
- It is built to do the best job it can in predicting on the test data.

WORKING WITH A DICHOTOMOUS OUTCOME VARIABLE

A DICHOTOMOUS OUTCOME

- We will now go back to the outpatient dataset and predict whether a patient arrives or is a no show.
- This is a dichotomous (2-levels) outcome and the tree is often called a “classification tree” in this case.

```
op_data = pd.read_csv("no_show_behavioral_20000.csv") # Read in data.
op_data['Age'] = pd.to_numeric(op_data['Age'],errors='coerce') # Clean up Age.
op_data = op_data.dropna() # Remove rows with missing values.

# Get the continous variables.
Xcts = op_data[['Age', 'Schedule lag', 'Minutes', 'Prior no show', 'Prior arrived']]
# "Binarize" the categorical variables.
Xcat = pd.get_dummies(op_data[['Sex', 'Marital status', 'Race', 'LanguageRecode',
                               'Department', 'Insurance type']],drop_first=True) # Categorical
  variables.
```

REVIEW THE FINISHED PREDICTION DATA FRAME

```
X = pd.merge(Xcts, Xcat,left_index=True, right_index=True) # Join the two data frames.
y = op_data['Status'] # Identify the y-variable.
print(X.head(5))
```

```
    Age  Schedule lag  Minutes  Prior no show  Prior arrived  Sex_MALE  \
0  52.0            24       45              0              0         1
1  49.0            35       10              0              3         0
2  37.0            46       45              0              0         0
3  31.0             3       20              1              0         0
4  62.0            30       15              0              2         1

   Marital status_LIFE PARTNER  Marital status_MARRIED  \
0                            0                       0
1                            0                       0
2                            0                       0
3                            0                       0
4                            0                       0

   Marital status_MARRIED OR CIVIL UNION  Marital status_SEPARATED  ...  \
```

PREPARE THE DATA PARTITION

- As before, we need to split the data into the training and testing datasets.

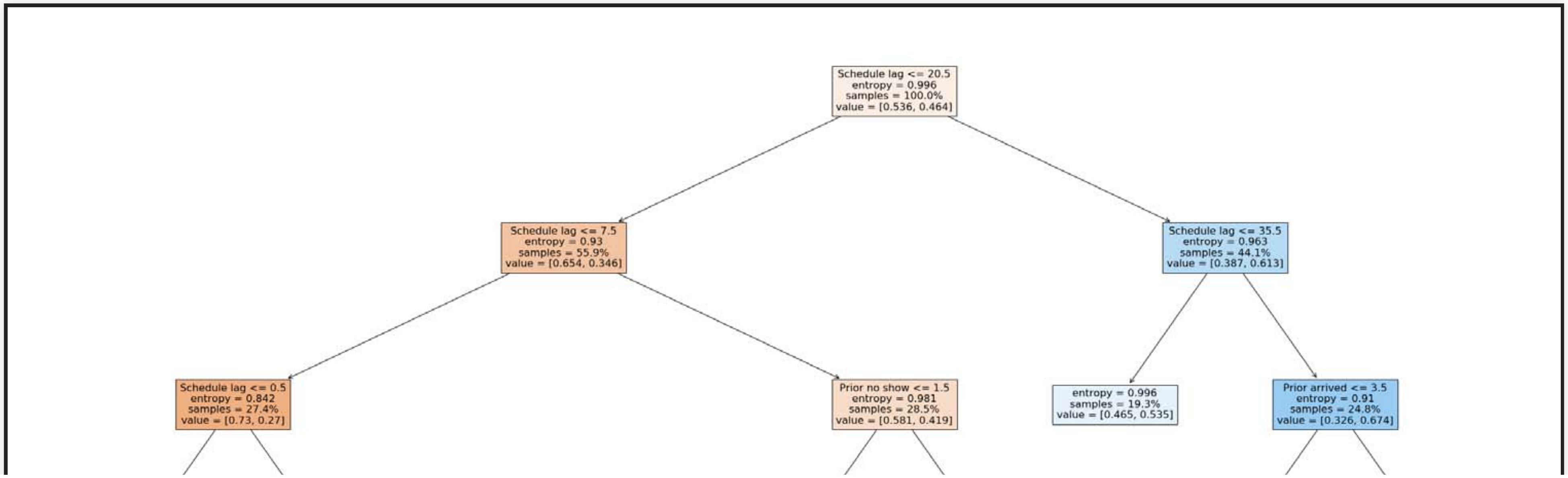
```
# Create the train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=1) # A 50/50
  split.
print(X_train.shape)
print(X_test.shape)
```

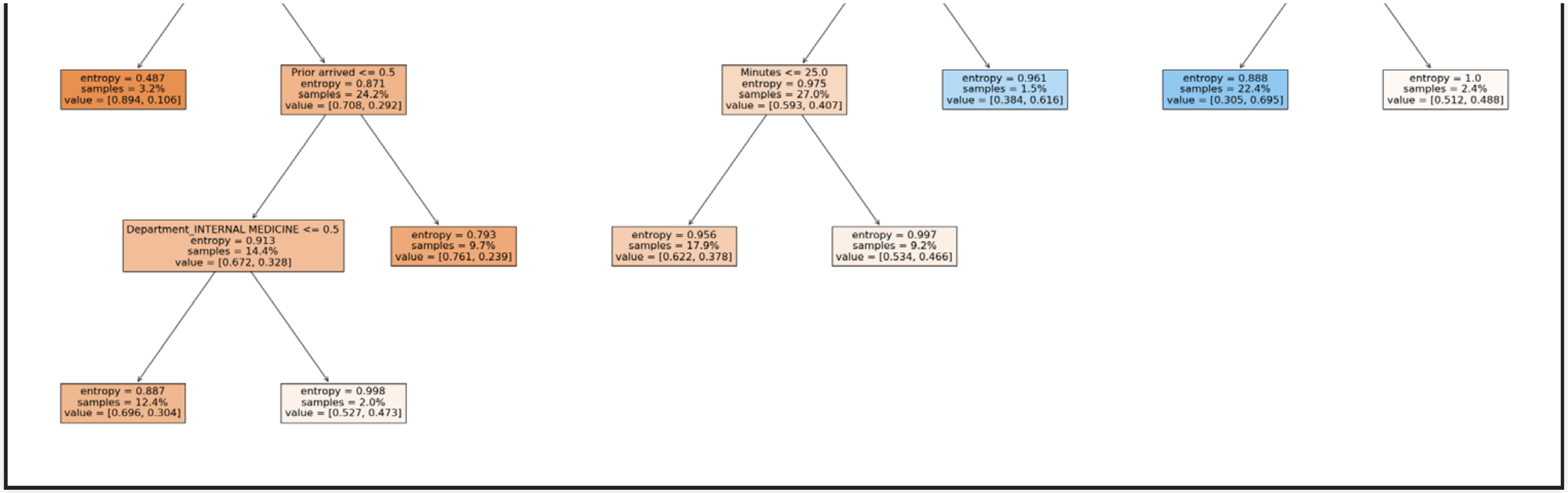
```
(9995, 66)
(9996, 66)
```

FIT THE TREE

- Notice that the tree is now an instance of “DecisionTreeClassifier”.
- Earlier, with continuous data, is was “DecisionTreeRegressor”.

```
clf = tree.DecisionTreeClassifier(criterion="entropy", max_leaf_nodes=10, random_state=0) # Set up the
  tree.
clf = clf.fit(X_train, y_train) # Fit the tree.
fig = plt.figure(num=None, figsize=(40, 25), dpi=80, facecolor='w', edgecolor='k')
tree.plot_tree(clf, filled=True, feature_names=X.columns,proportion=True);
```




entropy = 0.487
samples = 3.2%
value = [0.894, 0.106]
Prior arrived <= 0.5
entropy = 0.871
samples = 24.2%
value = [0.708, 0.292]
Minutes <= 25.0
entropy = 0.975
samples = 27.0%
value = [0.593, 0.407]
entropy = 0.961
samples = 1.5%
value = [0.384, 0.616]
entropy = 0.888
samples = 22.4%
value = [0.305, 0.695]
entropy = 1.0
samples = 2.4%
value = [0.512, 0.488]
Department_INTERNAL MEDICINE <= 0.5
entropy = 0.913
samples = 14.4%
value = [0.672, 0.328]
entropy = 0.793
samples = 9.7%
value = [0.761, 0.239]
entropy = 0.956
samples = 17.9%
value = [0.622, 0.378]
entropy = 0.997
samples = 9.2%
value = [0.534, 0.466]
entropy = 0.887
samples = 12.4%
value = [0.696, 0.304]
entropy = 0.998
samples = 2.0%
value = [0.527, 0.473]

VARIABLE IMPORTANCE

- It’s helpful to get a sense of variable importance.
- Schedule lag is so dominant here, almost nothing else matters.
- The two “behavioral variables”, do come second and third though!

```
importances =
  pd.DataFrame({'feature':X_train.columns,'importance':np.round(clf.feature_importances_,3)})
importances = importances.sort_values('importance',ascending=False)
print(importances[:5]) # Top 5 variables.
```

```
                              feature  importance
1                        Schedule lag       0.887
4                       Prior arrived       0.055
3                       Prior no show       0.022
42   Department_INTERNAL MEDICINE       0.019
2                             Minutes       0.017
```

MEASURES OF CLASSIFICATION ACCURACY

MEASURES OF CLASSIFICATION ACCURACY

- When the outcome variable is categorical the natural way to think about accuracy is whether the model's classification of an observation is correct.
- When the model is doing well, those cases that actually are positive are classified as positive.
- And those that are negative, are classified as negative.
- When the model is hopeless, you might as well be flipping a coin to classify.
- When an observation is incorrectly classified it is either a "False Positive", or a "False Negative".
- A "True Positive" is an actual positive that is classified as positive.
- A True Negative is an actual negative that is classified as a negative.

FALSE POSITIVES, FALSE NEGATIVES AND THE CONFUSION MATRIX

- A confusion matrix is a 2-by-2 table that cross-classifies the data by its true state against its predicted state.

		Classify Arrived	Classify No show	Total
True state	Arrived	3753	1756	5509
	No show	1918	2569	4487

- If we define No show as a "positive" then the false positive rate is 1756/5509 = 0.32.
- The false negative rate is 1918/4487 = 0.43.

OBTAINING THE CONFUSION MATRIX

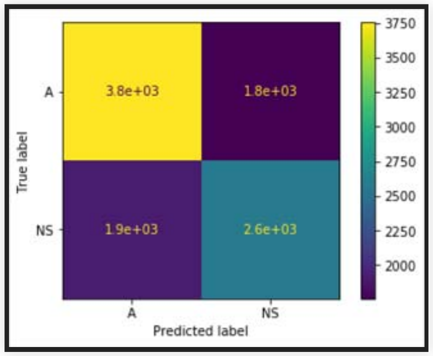
- sklearn has commands to obtain and plot the confusion matrix.
- By default it is calculated using a 50% cut-off rule: if the predicted probability of being a no show is greater than 50% then classify it as a no show otherwise classify it as an arrived.

```
# Summarize the fit.
# Confusion matrix.
from sklearn.metrics import confusion_matrix, plot_confusion_matrix
y_pred = clf.predict(X_test) # The predict method will classify the observation

cm = confusion_matrix(y_test, y_pred) # Cross-classify the "truth" against the "predictions".
print(cm)
```

```
[[3753 1756]
 [1918 2569]]
```

PLOTTING THE CONFUSION MATRIX

```
plot_confusion_matrix(clf, X_test, y_test);
```



USING A PROBABILITY CUT-OFF OTHER THAN 0.5

- Below we save the predicted probabilities and then create a classifier based on whether the predicted probability is greater than **0.25** .

```
y_prob_pred = clf.predict_proba(X_test) # The predict_proba method will return the predicted
  probability.
y_pred = np.where(y_prob_pred[:,1] > 0.25, "NS", "A") # From numpy, classify according to the predicted
  probability.
#y_pred = ["NS" if (x > 0.25)  else "A" for x in y_prob_pred[:,1]] # We could use a list comprehension
  as well.
cm = confusion_matrix(y_test, y_pred)
cm
```

```
array([[1014, 4495],
       [ 272, 4215]], dtype=int64)
```

AN ALTERNATIVE CONFUSION MATRIX

- This is the result of using 0.25 as the cut-off probability.

		Classify Arrived	Classify No show	Total
True state	Arrived	1014	4495	5509
	No show	272	4215	4487

- If we define No show as a “positive” then the false positive rate is 4495/5509 = 0.82.
- The false negative rate is 272/4487 = 0.05.

THE ROC CURVE

- Rather than deciding to use a single classification cut-off probability, we consider all possible cut-offs.
- For each cut-off there is a two-by-two confusion matrix and therefore a false positive and false negative calculation.
- The entire family of false positive, and false negative values are summarized through an ROC curve (Receiver Operating Characteristic).
- Formally the ROC curve plots the True Positive rate against the False Positive rate, for every possible probability cut-off value.

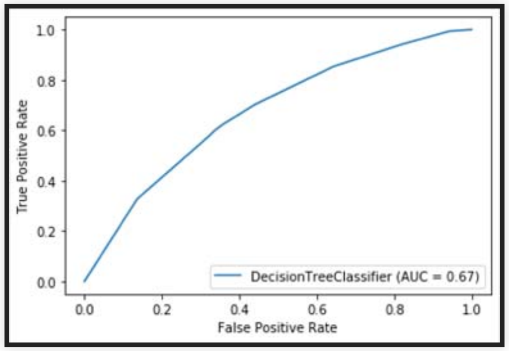
PLOTTING THE ROC CURVE

- Below we plot the ROC curve for the ***test*** data.
- A good classifier will have many true positives for each false positive.
- This means that a good ROC curves rises steeply, and one way to measure this is the Area Under the Curve (AUC).
- A perfect classifier has an AUC of 1. A useless classifier has an AUC of 0.5.
- AUC is usually used as a model comparison metric, where we prefer the model with the highest AUC on the test data.

CODE TO PLOT THE ROC CURVE

```
from sklearn.metrics import roc_curve, roc_auc_score, plot_roc_curve
y_pred = clf.predict_proba(X_test) # Get the predicted probablities.

plot_roc_curve(clf, X_test, y_test) # Plot the ROC curve.
print(roc_auc_score(y_test, y_pred[:, 1])) # Obtain the AUC.
```

```
0.6747078943656152
```



RANDOM FORESTS

RANDOM FORESTS

- A general criticism of trees is that their predictions are low bias, but unfortunately high variance.
- The classical way to reduce variability is to form an average.
- This is motivation for creating lots of trees, and averaging their individual predictions to get an *overall* prediction.
- This is the key idea in *ensemble learning* . That is, averaging predictions over a set of models.
- But how do we get different trees, if we only have one data set?
- The natural way to do this is to introduce randomness, and create sampled versions (bootstrapped/bagged) of the original data.
- Hence the name, *random forest* .
- Conventional wisdom says that the forest will almost always outperform a tree in prediction quality on the test data, but at the cost of interpretability.
- We tend to grow very deep trees when building a forest.

THE RANDOM FOREST

- The random forest will have all the tuning parameters of a tree plus new ones that define the nature of the forest. For example:
  - How many trees should be in the forest?
- Another aspect of the randomness in a tree used in a forest, is that when the algorithm considers a split, it doesn't look at all possible variables.
  - It ***randomly samples*** the variables, and considers a different random sample of variables at each and every split.
  - So each new tree is created by randomly sampling rows and repeatedly randomly sampling columns.
  - The idea behind using randomly sampled columns, is that the trees will be almost independent of one another and averaging is most effective at reducing variability, when the averaged elements are closer to independent.
  - Equivalently, if all the trees looked like one another, then averaging their predictions wouldn't make a difference, because they would all be the same.

SETTING UP A RANDOM FOREST

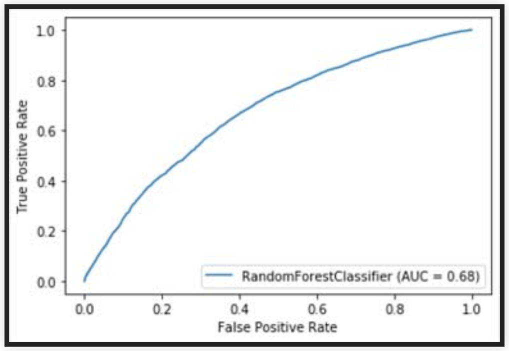
- The key command is ‘RandomForestClassifier’ which takes many arguments!

```
# %%timeit
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(n_estimators = 500,
                            criterion = "entropy",
                            max_depth=None,
                            max_features="auto",
                            bootstrap=True,
                            n_jobs=1,
                            random_state=0)
rf = rf.fit(X_train, y_train) # Fit the forest
```

ASSESS THE ACCURACY OF THE FOREST

```
y_pred = rf.predict_proba(X_test)

plot_roc_curve(rf, X_test, y_test)
print(roc_auc_score(y_test, y_pred[:, 1]))
```

```
0.6752625310779619
```



TUNING THE FOREST

- The choice of tuning parameters can sometimes make a big difference to the accuracy of a prediction algorithm.
- There can be many different tuning parameters in an algorithm, so changing them on an *ad hoc* basis is like shooting in the dark.
- We really need to set up a formal design where we search over a grid of tuning parameters.
- The ‘GridSearchCV’ function helps you do exactly that.
- It essentially sets up a computer experiment, over a grid of tuning parameters, looking for the best combination.

DESIGNING THE EXPERIMENT

- It is up to you to decide which parameters to tune.
- Create a dictionary with the keys as parameter names, and lists for their values.

```
from sklearn.model_selection import GridSearchCV
rf = RandomForestClassifier(n_estimators = 50,
                            criterion = "entropy",
                            max_depth=None,
                            max_features="auto",
                            bootstrap=True,
                            n_jobs=1,
                            random_state=0)

parameters = {'n_estimators':[50,100,150,200],'max_depth':[2,5,10,None], 'max_features':
  [1,5,10],'criterion':['entropy','gini']}
#parameters = {'n_estimators':[50,100],'max_depth':[2], 'max_features':[1], 'criterion':['entropy']}
parameters
```

```
{'n_estimators': [50, 100, 150, 200],
 'max_depth': [2, 5, 10, None],
 'max_features': [1, 5, 10],
 'criterion': ['entropy', 'gini']}
```

RUN THE GRID SEARCH

- Crossvalidation is used to assess the quality of each parameter combination.

```
clf = GridSearchCV(rf, parameters) # Run the computer experiment on the tuning parameters.
clf.fit(X_train, y_train)

clf.best_params_ # Have a look at the best parameter values.
```

```
{'criterion': 'entropy',
 'max_depth': 10,
 'max_features': 10,
 'n_estimators': 150}
```

GETTING AT THE RESULTS

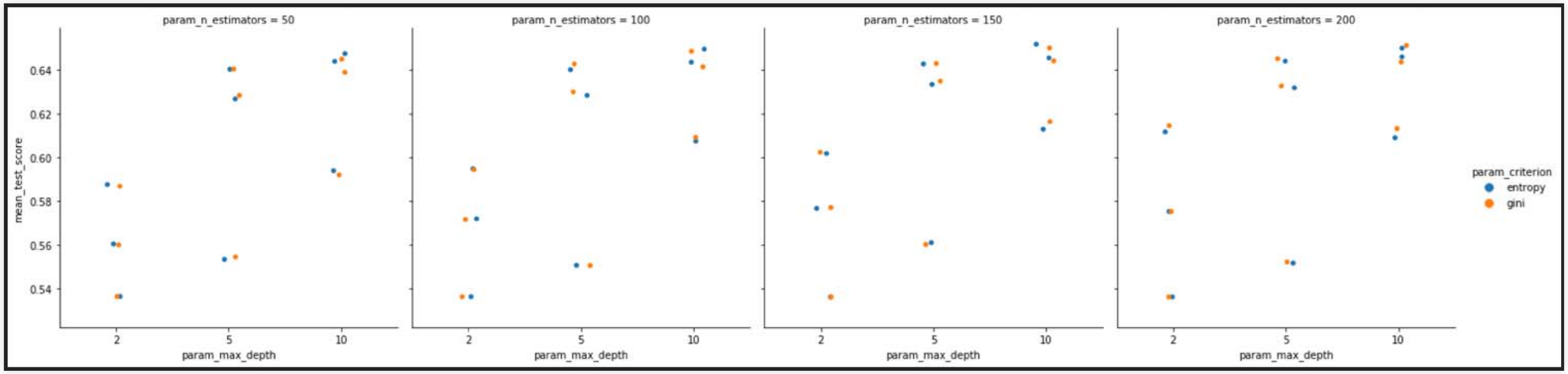
- The results of the experiment are stored and can be read in to a pandas data frame for ease of analysis and graphing.

```
exp_results = pd.DataFrame(clf.cv_results_) # This saves all the results into a pandas data frame.
print(exp_results.head(5))
```

```
   mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
0       0.083042      0.005117         0.013760        0.002253   
1       0.162440      0.007894         0.018720        0.006240   
2       0.242247      0.007348         0.024961        0.005134   
3       0.317218      0.009621         0.031202        0.000400   
4       0.096602      0.006721         0.011760        0.006043   

  param_criterion param_max_depth param_max_features param_n_estimators  \
0         entropy               2                  1                 50   
1         entropy               2                  1                100   
2         entropy               2                  1                150   
3         entropy               2                  1                200   
4         entropy               2                  5                 50   

                                              params  split0_test_score  \
```

PLOTTING THE RESULTS OF THE EXPERIMENT

```
sns.catplot(y='mean_test_score', x='param_max_depth', data=exp_results,
  col='param_n_estimators',hue='param_criterion');
```

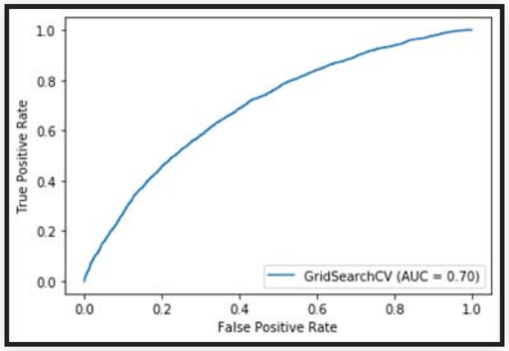


CHECKING THE TEST AUC

- Just as before we can obtain the test data set AUC.

```
y_pred = clf.predict_proba(X_test)
plot_roc_curve(clf, X_test, y_test);
print(roc_auc_score(y_test, y_pred[:, 1]))
```

```
0.6954931984588463
```



CONCLUSION

- The tuned random forest did have the best AUC.
- Practically speaking, there wasn’t much to be gained by the tuning.
- This is probably due to the fact that almost all the predictive power is in a single variable, Schedule Lag, and any model that includes it will do well.
- In practice, you never know what is to be gained by tuning the model, unless you try!

SUMMARY

SUMMARY

- Machine learning introduction.
- Decision trees:
  - Regression.
  - Classification
- Assessing the accuracy of a classifier.
- Ensemble learners – the random forest.

NEXT TIME

NEXT TIME

- Scientific computing with numpy.