DATA SOCIETY®

Introduction to classification - day 3

"One should look for what is and not what he thinks should be."
-Albert Einstein.

Module completion checklist

Objective	Complete
Discuss use cases for decision trees and random forests	
Summarize the concepts and math behind decision trees	
Transform the data so that we can use trees	
Implement the decision tree algorithm on a small subset	
Evaluate the model and store final results	
Implement decision tree on the entire dataset and evaluate its results	
Optimize the decision tree by tuning the hyperparameters	
Run the optimized model, predict and evaluate the new model	

Directory settings

- In order to maximize the efficiency of your workflow, you should encode your directory structure into `variables`
- Let the `main_dir` be the variable corresponding to your `af-werx` folder

```
# Set `main_dir` to the location of your `af-werx` folder (for Linux).
main_dir = "/home/[username]/Desktop/af-werx"
```

```
# Set `main_dir` to the location of your `af-werx` folder (for Mac).
main_dir = '/Users/[username]/Desktop/af-werx'
```

```
# Set `main_dir` to the location of your `af-werx` folder (for Windows).
main_dir = "C:\\Users\\[username]\\Desktop\\af-werx"
```

```
# Make `data_dir` from the `main_dir` and
# remainder of the path to data directory.
data_dir = main_dir + "/data"
```

```
# Do the same for your `plot_dir`.
plot_dir = main_dir + "/plots"
```

Loading packages

Let's load the packages we will be using:

```
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import pickle
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.model_selection import GridSearchCV

import graphviz
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score
from matplotlib.legend_handler import HandlerLine2D
```

- To install `graphviz` through Anaconda, run the following code in the terminal:

```
conda install graphviz
```

- Then, install `python-graphviz`, which is a Python library for `graphviz`

```
conda install python-graphviz
```

Working directory

- Set working directory to `data_dir`

```
# Set working directory.
os.chdir(data_dir)
```

```
# Check working directory.
print(os.getcwd())
```

```
/home/[user-name]/Desktop/af-werx/data
```

Introduction to Decision trees

- A decision tree is an effective modeling technique for classification problems
- This model that creates hierarchical solutions about the outcomes of a selected variable based on predictors
- It can be used for data exploration, making it an important step in model building
- Ultimately, it is a straightforward way of visualizing a decision

Decision trees: use cases

- Decision trees are **popular in a business setting** because they are intuitive
- Examples of decision tree use cases include loan approval, customer willingness to purchase an item in a particular setting (online vs offline) and forecasting future financial outcomes
- They are easy to **walk through and explain to stakeholders who are not familiar with data science**
- Today, we will walk through the conceptual explanation and then use a small subset to see a decision tree in action
- We will then use it on a larger dataset so that we can compare the performance of the decision tree to previous and future models

Decision trees: pros and cons

- Decision trees are **great** when used for:
  - Classification and regression
  - Handling numerical and categorical data
  - Handling data with missing values
  - Handling data with non-linear relationships between parameters

- Decision trees are **not very good** at:
  - Generalization: they are known for overfitting
  - Robustness: small variations in data can result in a different tree
  - Mitigating bias: if some classes dominate, trees may be unbalanced and biased

Goal for today

- Introduce and build a **decision tree** to predict vulnerable vs non-vulnerable households in our Costa Rican dataset
- Compare a decision tree to the methods we have learned so far
- Use a small subset to see a decision tree in action
- Use the larger `costa_subset` to build and optimize a decision tree model
- Add our final decision tree model score to the `model_final` dataset

Module completion checklist

Objective	Complete
Discuss use cases for decision trees and random forests	✔
Summarize the concepts and math behind decision trees	
Transform the data so that we can use trees	
Implement the decision tree algorithm on a small subset	
Evaluate the model and store final results	
Implement decision tree on the entire dataset and evaluate its results	
Optimize the decision tree by tuning the hyperparameters	
Run the optimized model, predict and evaluate the new model	

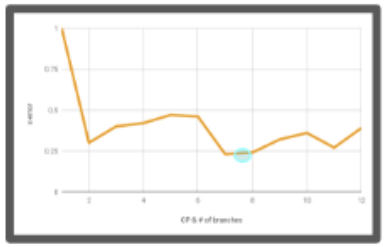
Understanding decision trees

- Decision trees will be your **good friend** when you are data mining
- They are **intuitive** and popular because they **provide explicit rules for classification** and **cope well with heterogeneous data, missing data, and nonlinear effects**
- Decision trees **predict** the target value of an item by **mapping observations about the item**

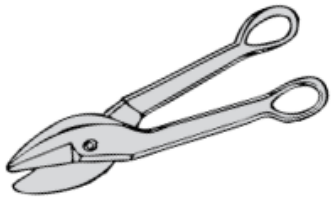
Decision trees: process

Step 1:
Grow tree on training data

Step 2:
Examine Model output



Step 3:
Prune Tree

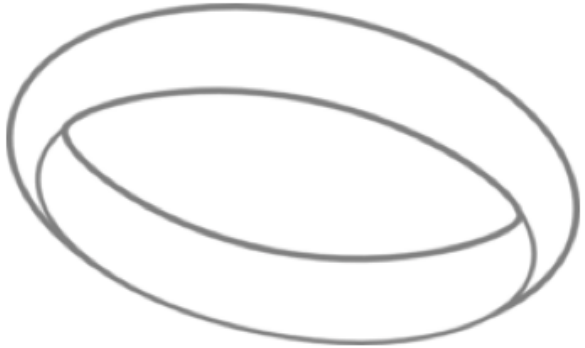


Step 4:
Check performance on test data

	Act +	Act -	
Pred +			
Pred -			

Finding the most important question

- Which question is more important on a date?
- **The question with the more relevant information**

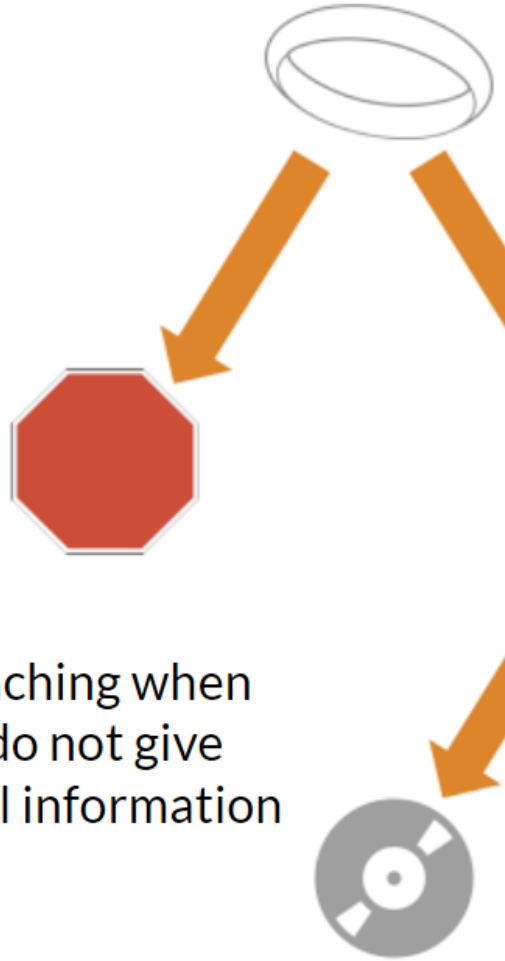


Are you married?

>

What music do you like?

Growing decision trees steps



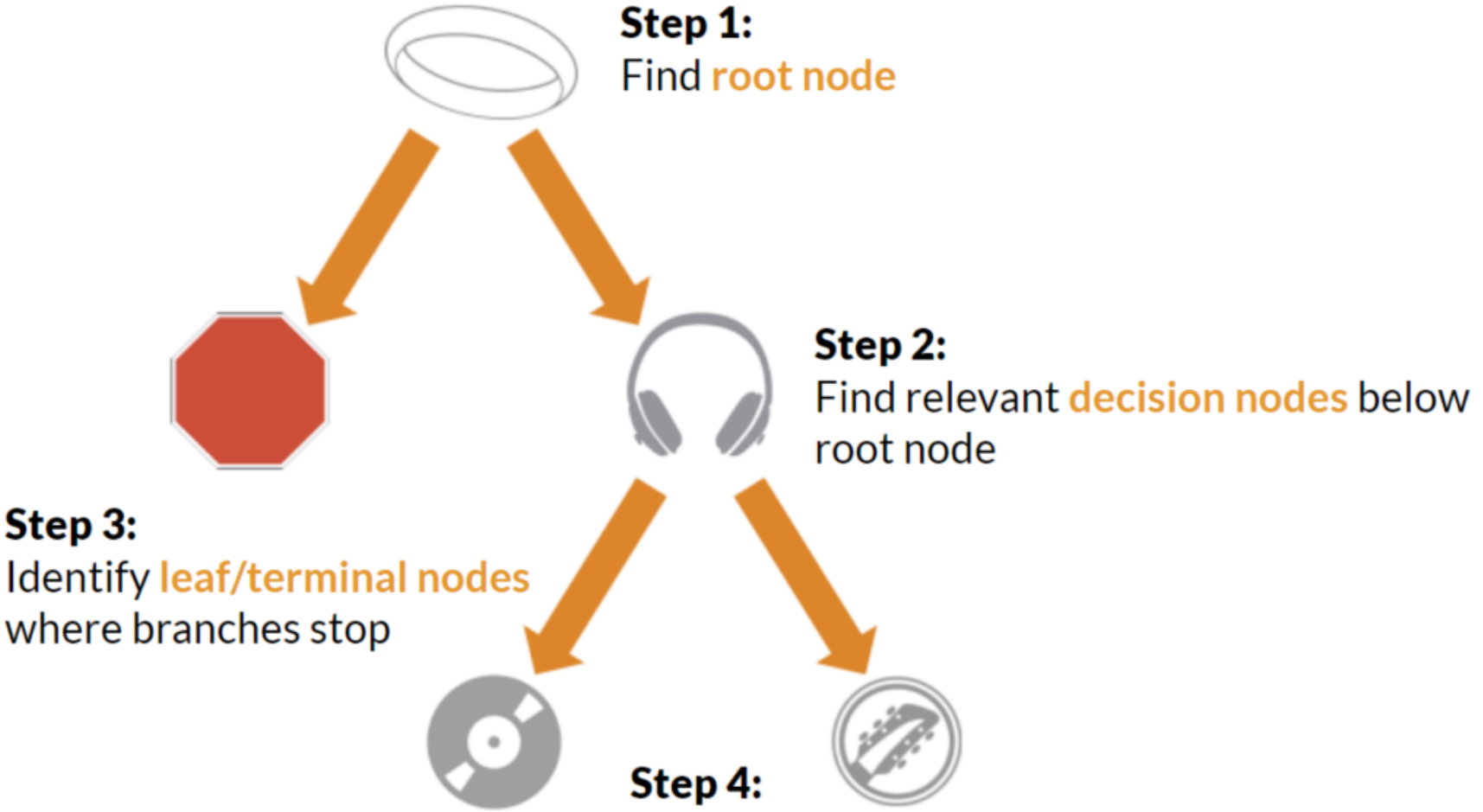
Step 1:
Ask the question with the most amount of information

Step 2:
Conditional on the first answer, select the next most important question

Step 3:
Stop branching when answers do not give additional information

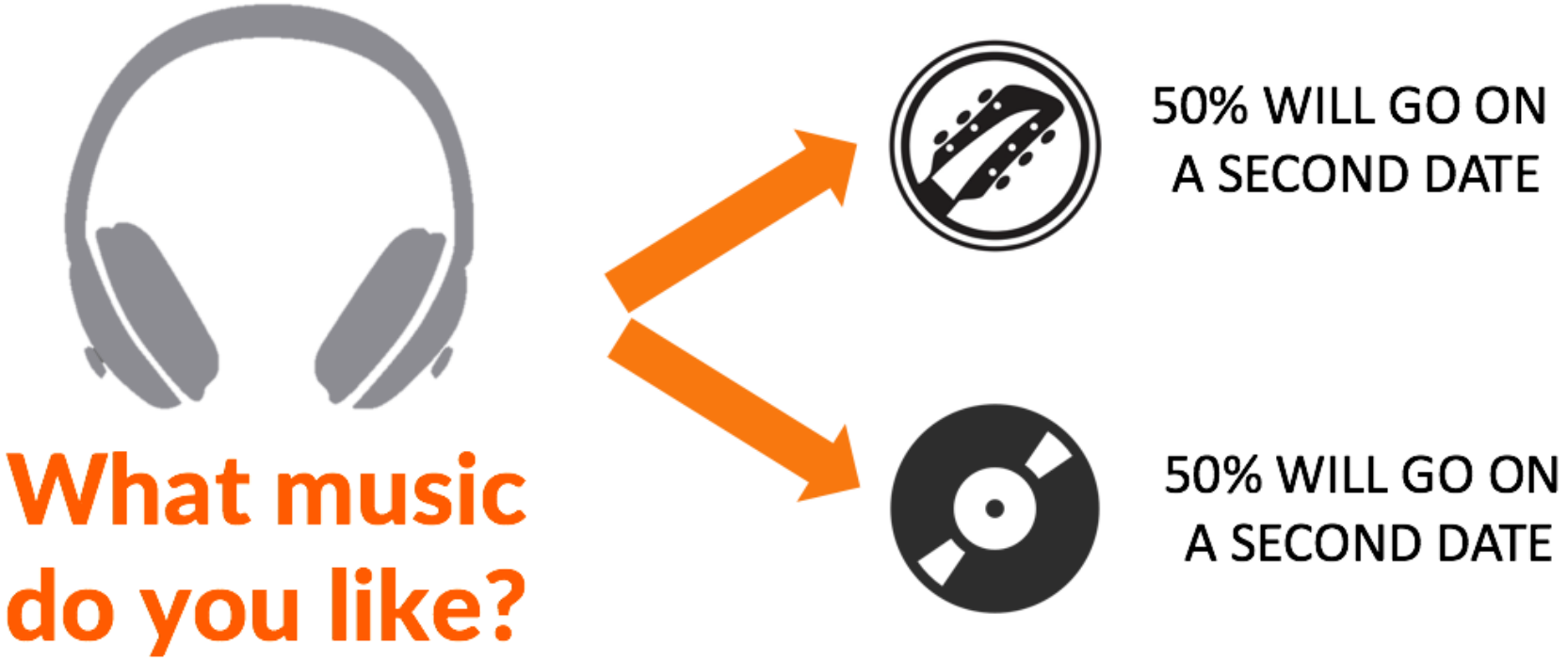
Step 4:
Repeat steps 2 and 3 for each question branch

Growing decision trees with vocabulary



When should you stop asking questions?

- **When the answer no longer provides additional relevant information**



The math of the splits

- How do we decide which node to split and how to split it?
- There are two **impurity** functions that are most commonly used with tree-based models
  - Gini
  - Entropy
- The `sklearn.tree` **algorithm** uses Gini, so this is the method we will focus on today

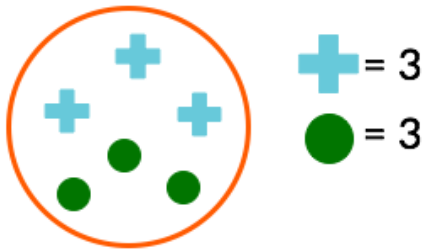
What does Gini do?

- Gini measures the **probability of misclassification** in the model for each branch. Gini ranges from 0 to 1.

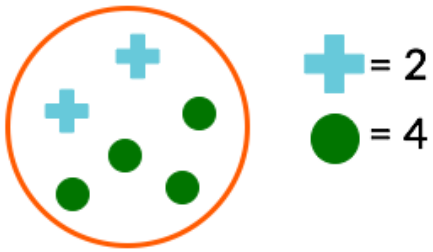
$$Gini(E) = 1 - \sum_{j=1}^{c} p_j^2$$

P*i* = the probability that a random selection would have state *i*

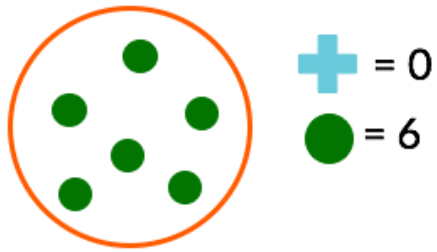
$$\text{Gini impurity} = 1 - \text{sum}[\ (P_i)^2\ ]$$



$1-(3/6)^2-(3/6)^2$



$1-(4/6)^2-(2/6)^2$



$1-(6/6)^2-(0/6)^2$

Knowledge Check 1

Module completion checklist

Objective	Complete
Discuss use cases for decision trees and random forests	✔
Summarize the concepts and math behind decision trees	✔
Transform the data so that we can use trees	
Implement the decision tree algorithm and visualize	
Evaluate the model and store final results	
Implement decision tree on the entire dataset and evaluate its results	
Optimize the decision tree by tuning the hyperparameters	
Run the optimized model, predict and evaluate the new model	

Datasets for trees

- **We will be using two datasets total, we discussed each of the datasets and use cases already**

- **One dataset to learn the concepts in class**
  - **Costa Rica household poverty data**

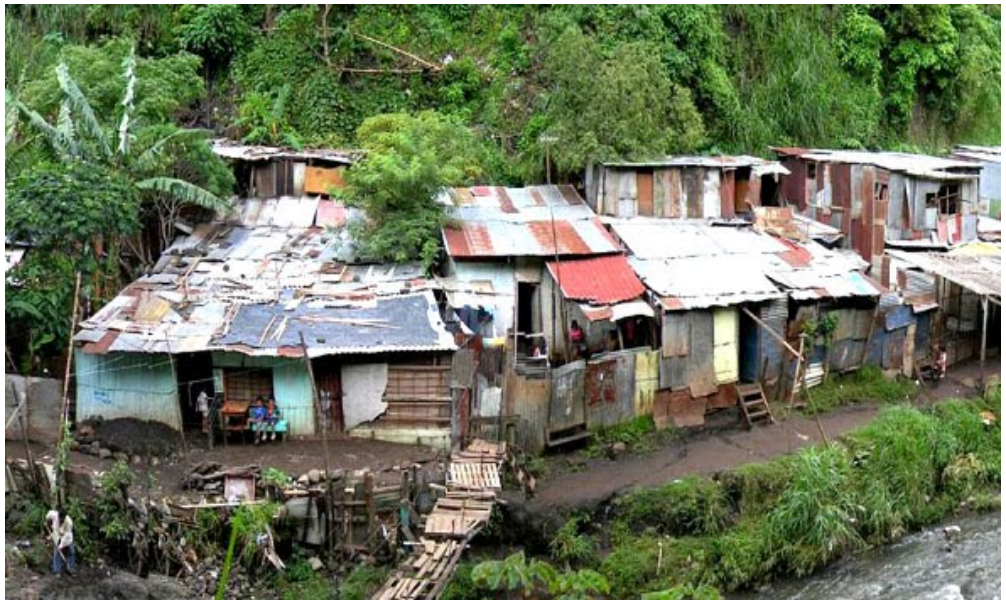
- **One dataset for our in-class exercises**
  - **Chicago census data**

Costa Rican poverty recap

- To improve on PMT, the IDB built a competition for Kaggle participants to use methods beyond traditional econometrics

- The given dataset contains Costa Rican household characteristics with a target of four categories:
  - extreme poverty
  - moderate poverty
  - vulnerable households
  - non vulnerable households

- Your manager would like for you to create a decision tree, using variables **number of rooms** and **number of adults**, to see if we can accurately predict poverty levels



Load the dataset

- Let's load the entire dataset

```
household_poverty = pd.read_csv("costa_rica_poverty.csv")
print(household_poverty.head())
```

```
  household_id        ind_id  rooms  ...  age  Target  monthly_rent
0     21eb7fcc1  ID_279628684      3  ...   43       4      190000.0
1     0e5d7a658  ID_f29eb3ddd      4  ...   67       4      135000.0
2     2c7317ea8  ID_68de51c94      8  ...   92       4           NaN
3     2b58d945f  ID_d671db89c      5  ...   17       4      180000.0
4     2b58d945f  ID_d56d6f5f5      5  ...   37       4      180000.0

[5 rows x 84 columns]
```

- The entire dataset consists of 9557 observations and 84 variables

Converting the target variable

- Let's convert poverty to a binary target variable, which will help to balance it out
- The levels translate to 1, 2, and 3 as being **vulnerable** households
- Level 4 is **non vulnerable**
- For this reason, we will convert all 1, 2 and 3 to `vulnerable` and 4 to `non_vulnerable`

```
household_poverty['Target'] = np.where(household_poverty['Target'] <= 3, 'vulnerable',
'non_vulnerable')
```

```
print(household_poverty['Target'].head())
```

```
0    non_vulnerable
1    non_vulnerable
2    non_vulnerable
3    non_vulnerable
4    non_vulnerable
Name: Target, dtype: object
```

Data prep: target

- The next step of our data cleanup is to ensure the target variable is binary and has a label
- Let's look at the `dtype` of `Target`

```
print(household_poverty.Target.dtypes)
```

```
object
```

- We want to convert this to `bool` so that it is a binary class

```
household_poverty["Target"] = np.where(household_poverty["Target"] == "non_vulnerable", True, False)

# Check class again.
print(household_poverty.Target.dtypes)
```

```
bool
```

Subsetting data

- In this module, we will once again subset the dataset
- We don't want to use `monthly_rent` as a variable right now because we had so many NAs
- We want to see if maybe the **number of rooms** and **number of adults** would predict poverty level well
- Then, we are going to predict the same with a larger subset

Subsetting data

- Let's subset our data so that we have the variables we need for building our model
- Let's name this subset `costa_tree_subset`

```
costa_tree_subset = household_poverty[['rooms', 'num_adults', 'Target']]
print(costa_tree_subset.head())
```

```
   rooms  num_adults  Target
0      3           1    True
1      4           1    True
2      8           1    True
3      5           2    True
4      5           2    True
```

- For now, we are only going to use `rooms` and `num_adults` for a simple decision tree

Summarize the data

- Let's use the `.describe()` function within `pandas` to summarize our data

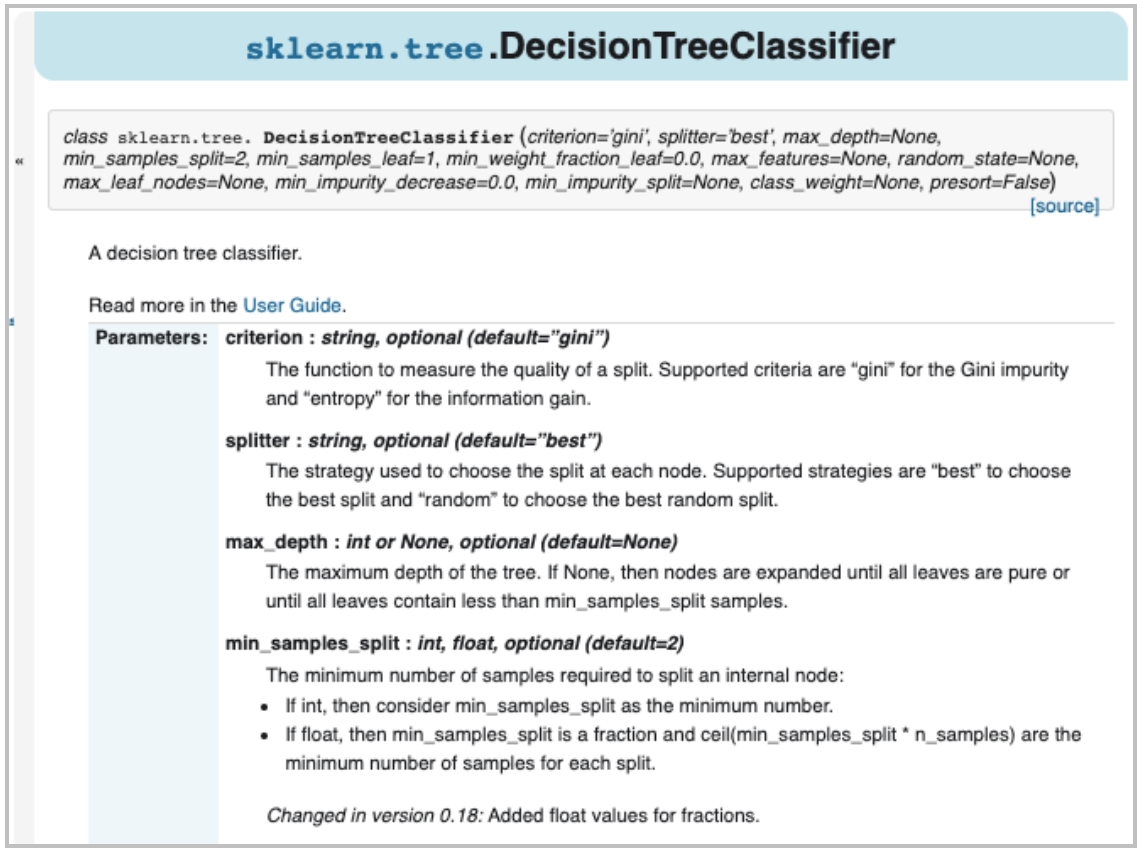
```
print(costa_tree_subset.describe())
```

```
              rooms   num_adults
count  9557.000000  9557.000000
mean      4.955530     2.592445
std       1.468381     1.166074
min       1.000000     0.000000
25%       4.000000     2.000000
50%       5.000000     2.000000
75%       6.000000     3.000000
max      11.000000     9.000000
```

- What do you notice about the variables we're about to use?

scikit-learn: tree

- We will be using the `DecisionTreeClassifier` library from `scikit-learn`



sklearn.tree.DecisionTreeClassifier

class sklearn.tree. **DecisionTreeClassifier** (*criterion='gini', splitter='best', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None, random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, class_weight=None, presort=False*) [source]

A decision tree classifier.

Read more in the User Guide.

Parameters:

criterion : *string, optional (default="gini")*

The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.

splitter : *string, optional (default="best")*

The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.

max_depth : *int or None, optional (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 : *int, float, optional (default=2)*

The minimum number of samples required to split an internal node:

- If int, then consider min_samples_split as the minimum number.
- If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

- For all the parameters of the `tree` package, visit ***scikit-learn's documentation***

Pre-processing steps

- Now that we know the problem and have our `costa_tree_subset` dataset loaded, it's time to get the data ready to model
- Let's start with data pre-processing
  - We don't need to scale since trees are not sensitive to unscaled data
  - We don't need to look at the number of NAs because we already did this and have loaded the cleaned dataset
- **However, on your own data, these are steps you must walk through before modeling**

Using a sub-sample

- One thing we will do today is run our tree on a sub-sample
- This is so we can clearly see our splits when we visualize the tree
- Then, we will run the tree on the entire `household_poverty` dataset and record our results
- Finally, we will discuss a few ways to optimize the decision tree and re-run our optimized model

Module completion checklist

Objective	Complete
Discuss use cases for decision trees and random forests	✔
Summarize the concepts and math behind decision trees	✔
Transform the data so that we can use trees	✔
Implement the decision tree algorithm and predict on test	
Evaluate the model, discuss ways to optimize and store final results	
Implement decision tree on the entire dataset and evaluate its results	
Optimize the decision tree by tuning the hyperparameters	
Run the optimized model, predict and evaluate the new model	

Splitting our subset

- Let's split our dataset to create our training and test sets `costa_tree_subset` into `X_small` and `y_small`
- `X_small` will consist of the first 60 rows of the two predictors, `rooms` and `num_adults`
- `y_small` will consist of the first 60 rows of the target variable, `Target`

```
np.random.seed(1)
X_small = costa_tree_subset.iloc[0:60, 0:2]
y_small = costa_tree_subset.iloc[0:60, 2:3]
```

Tree: run on small dataset

- Now let's run our tree on the entire `X_small` dataset

```
# Implement the decision tree on X_small.
clf = tree.DecisionTreeClassifier()
clf_fit_small = clf.fit(X_small, y_small)

# Look at our generated model:
print(clf_fit_small)
```

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
                       max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort=False,
                       random_state=None, splitter='best')
```

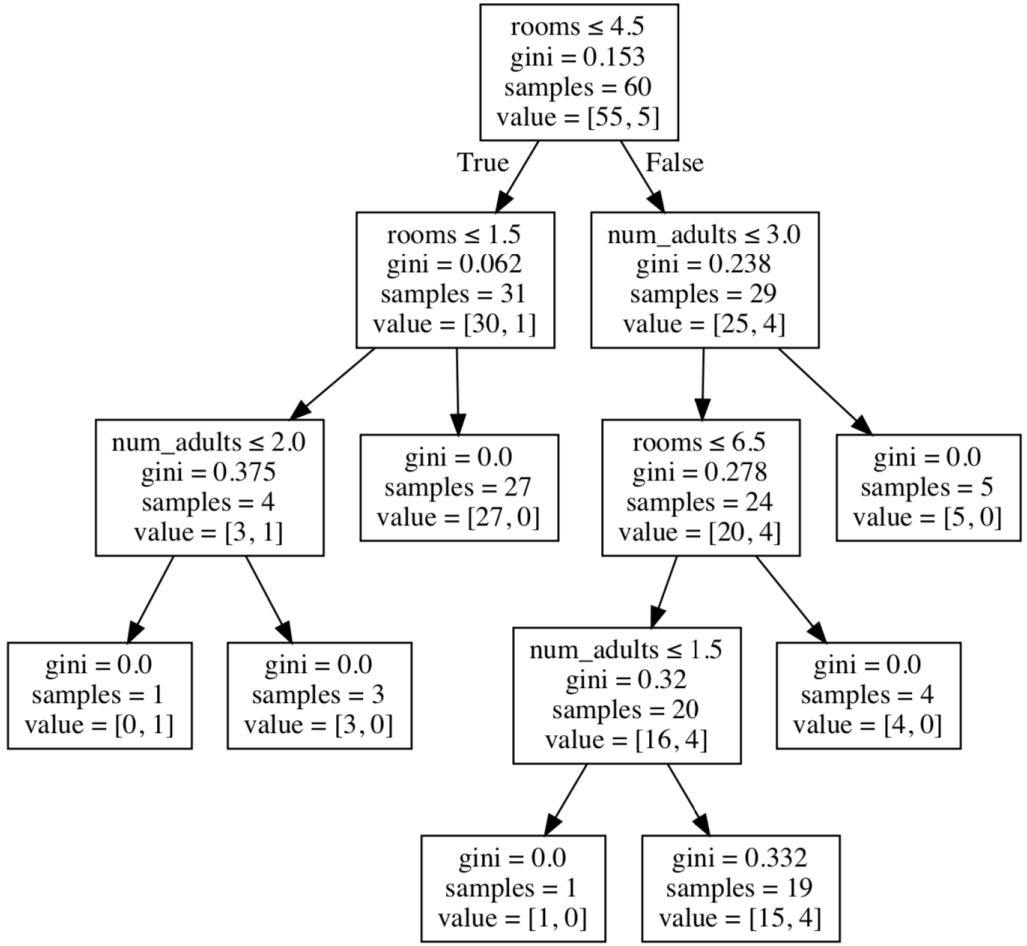
Visualize: graphviz

- Your manager wants you to visualize your decision tree in order to show your colleagues for poverty level determination
- We can visualize trees by using `graphviz`
- You have to ***install `graphviz` first, through Anaconda***
- We will use it today to illustrate the small tree we are building
- If you do have the package installed, you can run the code below!
- Otherwise, you will have time to try later and just follow along for now

```
# Change working directory to plot directory
os.chdir(plot_dir)

# Visualize `clf_fit_small`.
feature_cols = ['rooms','num_adults']
dot_data = tree.export_graphviz(clf_fit_small,
                                feature_names = feature_cols,
                                out_file = None)
graph = graphviz.Source(dot_data)
graph.render("trained_tree")
```

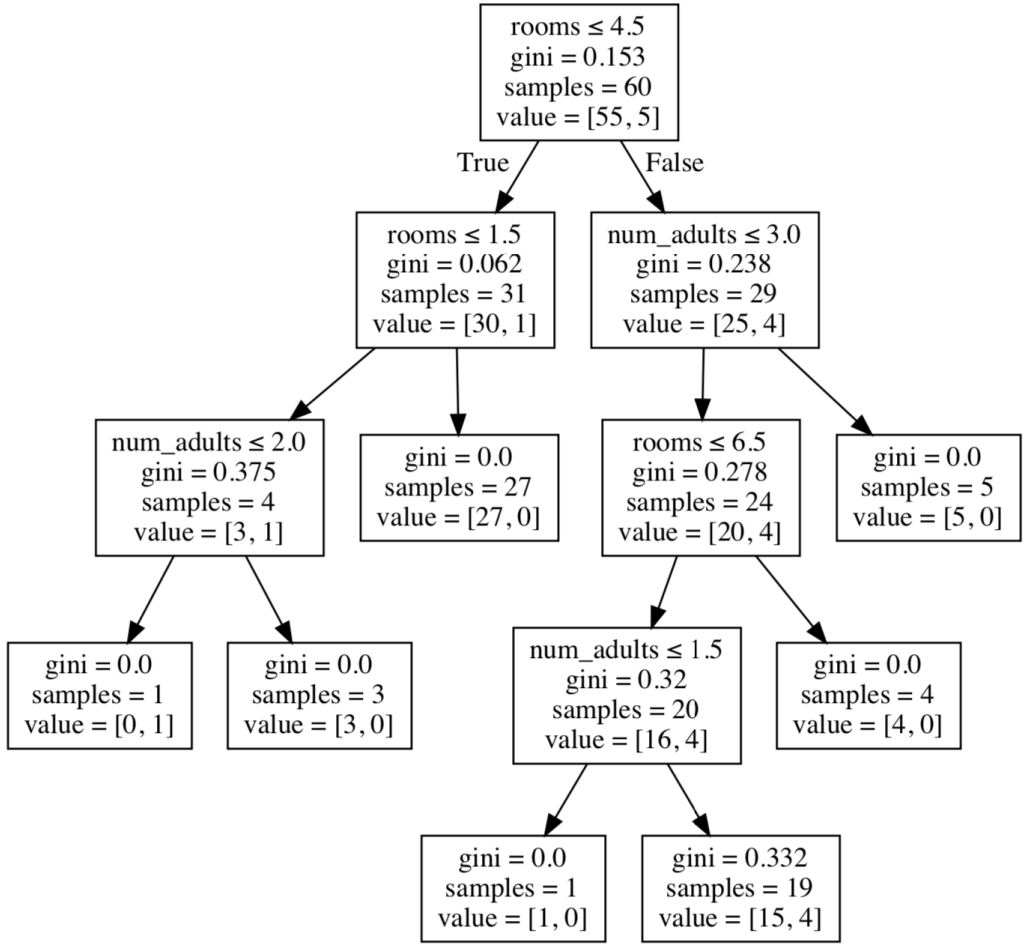
Visualize: graphviz

- Let's look at the **root node** to understand how to interpret output:
- `X[0] <= 4.5` :
  - All samples where `rooms` is `<= 4.5` go to the left child
  - All other samples go to the right child
- `gini = .153` : the Gini impurity of the node,
  - It describes how pure or mixed up classes are
  - Here, `.153` means that the classes are unevenly split
  - If it were closer to .5, then both classes would be evenly split



Visualize: graphviz

- `samples = 60`:
  - This means that the node 'contains' 60 samples
  - The tree was trained on 60 samples, which is what our dataset consisted of
- `value = [5,55]`:
  - The classifications of `Target`
  - 5 samples to `low_value` and 55 to `high_value`

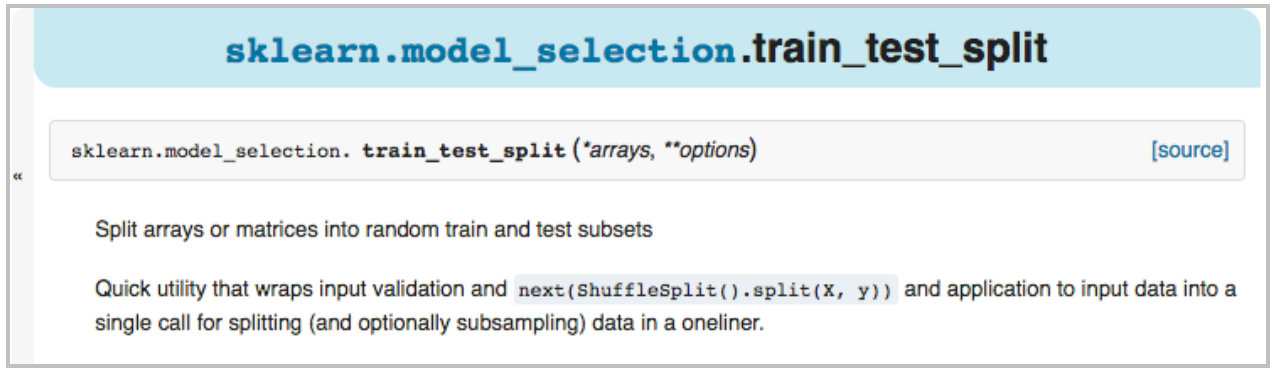


Tree: costa_tree

- Now, let's run the tree on the full `costa_tree_subset` dataset
- Remember, this dataset still only has **two predictors and one target class**
- After we predict on this small dataset, we are going to run a decision tree on the entire `household_poverty`
- **We will evaluate both models and add them to our `model_final` dataframe**

scikit-learn: train_test_split

- We will be using the `train_test_split` library from `scikit-learn`
- You should be familiar with this by function by now, but here is a refresher:



sklearn.model_selection.train_test_split

sklearn.model_selection. **train_test_split** (**arrays*, ***options*) [source]

Split arrays or matrices into random train and test subsets

Quick utility that wraps input validation and `next(ShuffleSplit().split(X, y))` and application to input data into a single call for splitting (and optionally subsampling) data in a oneliner.

- Inputs are:
  - Lists, numpy arrays, scipy-sparse matrices, or pandas dataframes
- For all the parameters of the `tree` package, visit ***scikit-learn's documentation***

Split into train and test

- Split the cleaned data into a training set and test set
- Remember, we already imported the given package at the start of this lesson
- Otherwise, we would have to import it now

```
# Change working directory back to data directory.
os.chdir(data_dir)

# Split into train and test, use sklearn.
# Create training and test vars.
np.random.seed(1)
X = costa_tree_subset[['rooms','num_adults']]
y = np.array(costa_tree_subset['Target'])

# Split into train and test.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

print(X_train.shape, y_train.shape)
```

```
(6689, 2) (6689,)
```

```
print(X_test.shape, y_test.shape)
```

```
(2868, 2) (2868,)
```

Fit decision tree and predict

- Now we will run `tree` on `X_train` and `y_train` and then predict on `X_test`

```
# Implement the decision tree on X_train.
clf = tree.DecisionTreeClassifier()
clf_fit = clf.fit(X_train, y_train)

# Predict on X_test.
y_predict = clf_fit.predict(X_test)
```

Module completion checklist

Objective	Complete
Discuss use cases for decision trees and random forests	✔
Summarize the concepts and math behind decision trees	✔
Transform the data so that we can use trees	✔
Implement the decision tree algorithm and predict on test	✔
Evaluate the model and store final results	
Implement decision tree on the entire dataset and evaluate its results	
Optimize the decision tree by tuning the hyperparameters	
Run the optimized model, predict and evaluate the new model	

Evaluate model

- By now, we are familiar with classification metrics for evaluating a model
- We can **use the same metrics we have discussed so far** to measure how well our simple decision tree is doing
  - Accuracy
  - Confusion matrix
  - AUC
  - ROC
- Let's calculate the confusion matrix and accuracy:

```
# Accuracy score.
acc_score = accuracy_score(y_test, y_predict)
print(acc_score)
```

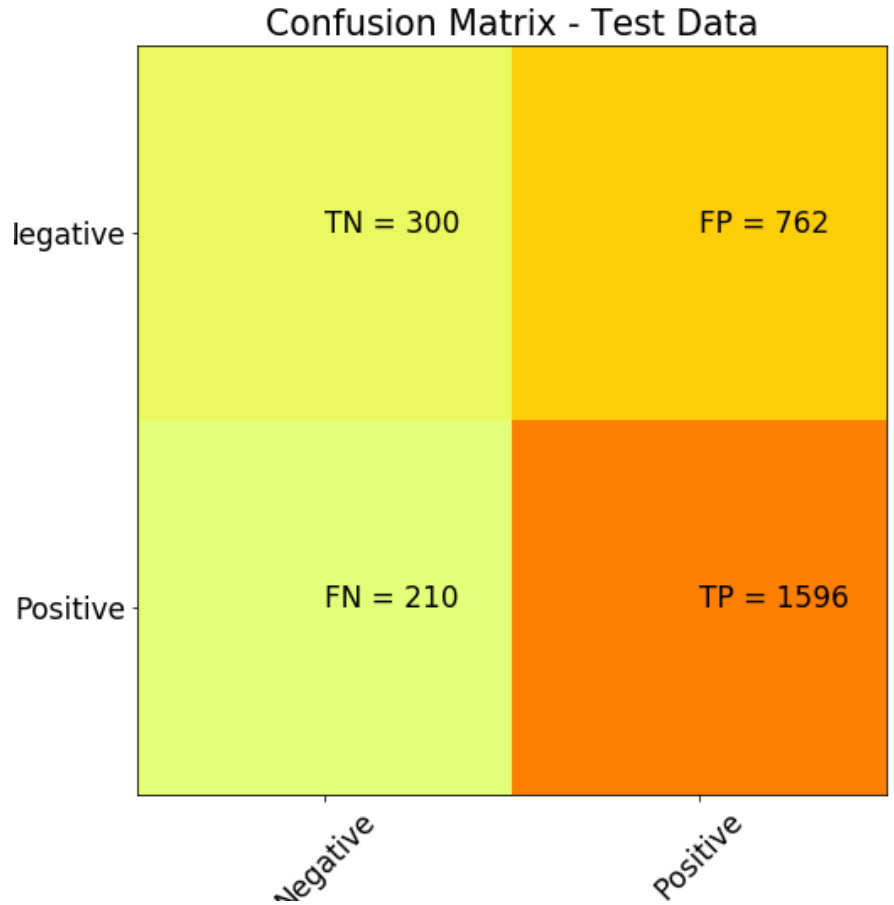
```
0.6610878661087866
```

```
# Confusion matrix for first model.
cm_tree = confusion_matrix(y_test, y_predict)
```

Plot confusion matrix

- Let's plot our confusion matrix

```
plt.clf()
plt.imshow(cm_tree, interpolation='nearest',
cmap=plt.cm.Wistia)
classNames = ['Negative','Positive']
plt.title('Confusion Matrix - Test Data')
plt.ylabel('True label')
plt.xlabel('Predicted label')
tick_marks = np.arange(len(classNames))
plt.xticks(tick_marks, classNames, rotation=45)
plt.yticks(tick_marks, classNames)
s = [['TN','FP'], ['FN', 'TP']]
for i in range(2):
    for j in range(2):
        plt.text(j,i, str(s[i][j]) + " = " +
str(cm_tree[i][j]))
plt.show()
```



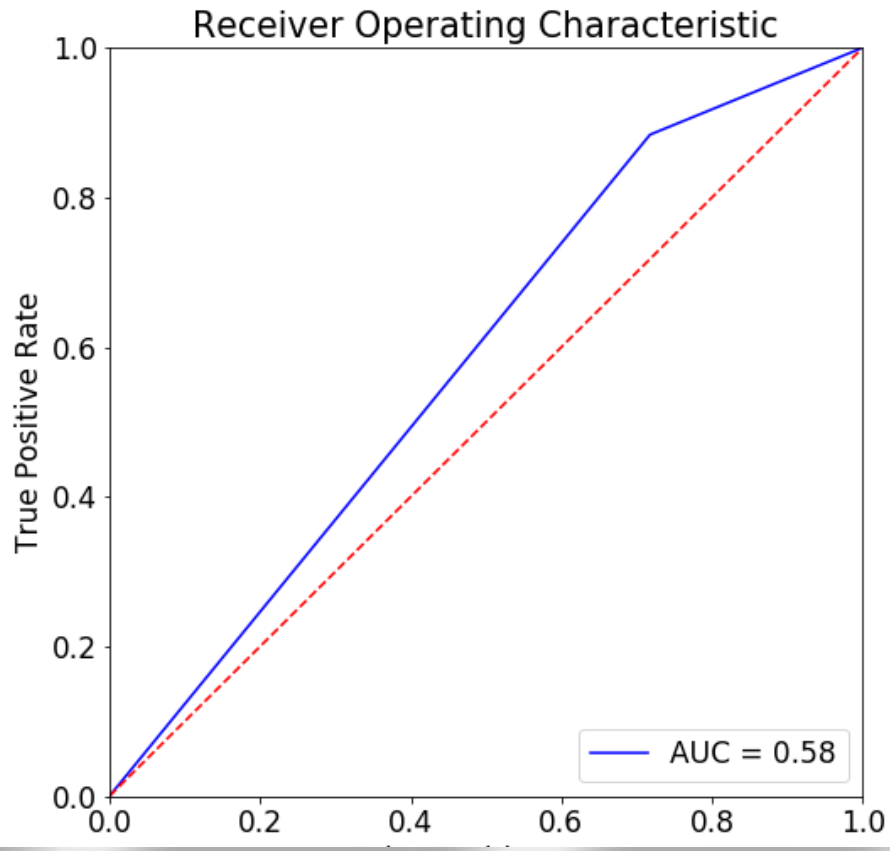
Plot ROC and calculate AUC

- Finally, let's plot our ROC curve and calculate AUC

```
# Calculate metrics for ROC (fpr, tpr) and
calculate AUC.
fpr, tpr, threshold = metrics.roc_curve(y_test,
y_predict)
roc_auc = metrics.auc(fpr, tpr)

# Plot ROC.
plt.title('Receiver Operating Characteristic')
plt.plot(fpr, tpr, 'b', label = 'AUC = %0.2f' %
roc_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```

```
(0, 1)
```

```
(0, 1)
```



Save metrics to model_final dataframe

```
model_final_tree = pickle.load(open("model_final_logistic.sav","rb"))
```

```
# Add this final model champion to our dataframe.
model_final_tree = model_final_tree.append({'metrics' : "accuracy" ,
                                            'values' : round(acc_score,4),
                                            'model':'tree_simple_subset' } ,
                                            ignore_index = True)
print(model_final_tree)
```

```
    metrics  values                   model
0  accuracy  0.6046                   knn_5
1  accuracy  0.6188       knn_GridSearchCV
2  accuracy  0.6287                  knn_29
3  accuracy  0.6356                logistic
4  accuracy  0.7845  logistic_whole_dataset
5  accuracy  0.7859          logistic_tuned
6  accuracy  0.6611     tree_simple_subset
```

Knowledge Check 2

Exercise 1

Module completion checklist

Objective	Complete
Discuss use cases for decision trees and random forests	✔
Summarize the concepts and math behind decision trees	✔
Transform the data so that we can use trees	✔
Implement the decision tree algorithm and predict on test	✔
Evaluate the model and store final results	✔
Implement decision tree on the entire dataset and evaluate its results	
Optimize the decision tree by tuning the hyperparameters	
Run the optimized model, predict and evaluate the new model	

View the dataset

- Let's run the model on the whole `household_poverty` dataset
- We have removed `monthly_rent` as discussed before, along with `ID` variables
- Save as `costa_tree`

```
costa_tree = household_poverty.drop(['household_id', 'ind_id', 'monthly_rent'], axis = 1)
print(costa_tree.head())
```

```
   rooms  tablet  males_under_12  ...  rural_zone  age  Target
0      3       0               0  ...           0   43    True
1      4       1               0  ...           0   67    True
2      8       0               0  ...           0   92    True
3      5       1               0  ...           0   17    True
4      5       1               0  ...           0   37    True

[5 rows x 81 columns]
```

Split into train and test set

- As we did previously, we split our data into training and test sets
- Let's run a decision tree initially on the training data

```
# Split the predictors from data.
X = costa_tree.drop('Target', axis = 1)
```

```
# Separate target from data.
y = np.array(costa_tree['Target'])
```

```
# Set the seed.
np.random.seed(1)
# Split data into training and test set, use a 70 train - 30 test split.
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size = .3)
```

Decision tree: build

- Let's build our decision tree and use all default parameters for now as our baseline model

```
# Set up decision tree model.
clf = tree.DecisionTreeClassifier()
print(clf)
```

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
                       max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort=False,
                       random_state=None, splitter='best')
```

- We can see that the default model contains many parameters, including:
  - `max_depth = None`
  - `min_samples_split = 2`
  - `min_samples_leaf = 1`
  - `max_features = None`
- Your manager would like for you to get into the final evaluation of the model and then optimize it, focusing on the parameters above

Decision tree: fit

- We fit the decision tree with `X_train` and `y_train`
- We will run the model on our training data and predict on test data

```
# Fit the model.
clf_fit = clf.fit(X_train, y_train)
```

Decision tree: predict

- We will predict on the test data using our trained model
- The result is a **vector of the predictions**

```
# Predict on X_test.
y_predict = clf_fit.predict(X_test)
print(y_predict)
```

```
[ True  True False ...  True  True False]
```

Decision tree: accuracy

- Let's calculate the accuracy of our tree and save it to our `model_final` dataframe

```
# Compute test model accuracy score.
tree_accuracy_score = metrics.accuracy_score(y_test, y_predict)
print("Accuracy on test data: ", tree_accuracy_score)
```

```
Accuracy on test data:  0.9407252440725244
```

- The decision tree has roughly a 20% increase in accuracy from the logistic model!
- **Is this result accurate?**
- The high increase in accuracy could be due to a multitude of reasons:
  - The classifier could be overfitting the dataset
  - The tree could be biased to classes which have a majority in the dataset
  - The training set and test set could also be very similar

Decision tree: accuracy

- Add the accuracy score to our `model_final` dataframe to keep track

```
# Add this model to our model champion dataframe.
model_final_tree = model_final_tree.append({'metrics' : "accuracy" ,
                                            'values' : round(tree_accuracy_score,4),
                                            'model':'tree_all_variables' } ,
                                            ignore_index = True)
print(model_final_tree)
```

```
    metrics  values                   model
0  accuracy  0.6046                   knn_5
1  accuracy  0.6188       knn_GridSearchCV
2  accuracy  0.6287                  knn_29
3  accuracy  0.6356                logistic
4  accuracy  0.7845  logistic_whole_dataset
5  accuracy  0.7859         logistic_tuned
6  accuracy  0.6611     tree_simple_subset
7  accuracy  0.9407     tree_all_variables
```

- Let's run a **10-fold cross-validation** to see if the results remain accurate

Cross-validation

- The input is an estimator, X, y and the number of folds for cross-validation
- It returns an **array of scores of the estimator for each run of the cross-validation**

sklearn.model_selection.cross_val_score

`sklearn.model_selection.` **`cross_val_score`**(*estimator, X, y=None, groups=None, scoring=None, cv='warn', n_jobs=None, verbose=0, fit_params=None, pre_dispatch='2*n_jobs', error_score='raise-deprecating'*) [source]

Evaluate a score by cross-validation

Read more in the User Guide.

Parameters:

estimator : *estimator object implementing 'fit'*
The object to use to fit the data.

X : *array-like*
The data to fit. Can be for example a list, or an array.

y : *array-like, optional, default: None*
The target variable to try to predict in the case of supervised learning.

groups : *array-like, with shape (n_samples,), optional*
Group labels for the samples used while splitting the dataset into train/test set.

scoring : *string, callable or None, optional, default: None*
A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)` which should return only a single value.

Similar to `cross_validate` but only a single metric is permitted.

If None, the estimator's default scorer (if available) is used.

cv : *int, cross-validation generator or an iterable, optional*
Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross validation,
- integer, to specify the number of folds in a `(Stratified)KFold`,
- CV splitter,
- An iterable yielding (train, test) splits as arrays of indices.

Cross-validation

- We can see that the scores are roughly between `0.6` to `0.7`
- So we can see that the model has overfit to the dataset, which means it won't be useful for other datasets

```
X = costa_tree.drop('Target', axis = 1)
y = np.array(costa_tree['Target'])
clf = tree.DecisionTreeClassifier()
print(cross_val_score(clf, X, y, cv = 10))
```

```
[0.67816092 0.63179916 0.68410042 0.67468619 0.67782427 0.65376569
 0.64502618 0.35183246 0.52879581 0.62931937]
```

- Let's remove the highly correlated variables from the dataset to improve accuracy

Highly-correlated variables

- First, we check the shape of the dataset

```
print(costa_tree.shape)
```

```
(9557, 81)
```

- We do not check for correlation between a predictor and the target variable, so let's drop `Target`

```
costa_hc = costa_tree.drop('Target', axis = 1)
```

Highly-correlated variables

- **Let's create a correlation matrix and drop the variables which have correlation higher than `0.7`** to improve accuracy

```
# Create correlation matrix.
corr_matrix = costa_hc.corr().abs()
```

```
# Select upper triangle of correlation matrix.
upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(np.bool))
```

```
# Find features with correlation greater than 0.7.
to_drop = [column for column in upper.columns if any(upper[column] > 0.7)]
print(to_drop)
```

```
['males_tot', 'females_tot', 'ppl_under_12', 'ppl_over_12', 'ppl_total', 'floor_cement',
'electric_coop', 'toilet_septic', 'cookenergy_gas', 'trash_burn', 'wall_good', 'roof_good',
'floor_good', 'female', 'num_child', 'num_adults', 'num_hh_total', 'educ_none', 'bedrooms',
'rural_zone']
```

```
# Drop the features we identified.
costa_tree = costa_tree.drop(costa_tree[to_drop], axis=1)
print(costa_tree.shape)
```

```
(9557, 61)
```

Highly-correlated variables

- Let's see the scores again by running 10-fold cross-validation on the reduced dataset

```
X = costa_tree.drop('Target', axis = 1)
y = np.array(costa_tree['Target'])
clf = tree.DecisionTreeClassifier()
print(cross_val_score(clf, X, y, cv = 10))
```

```
[0.70114943 0.74476987 0.70606695 0.62552301 0.7123431  0.62343096
 0.61989529 0.43246073 0.50890052 0.69424084]
```

- The scores have improved! Let's pickle this dataframe for use in future sessions

```
pickle.dump(costa_tree, open("costa_no_hc.sav","wb" ))
```

Knowledge check 3

Exercise 2

Module completion checklist

Objective	Complete
Discuss use cases for decision trees and random forests	✔
Summarize the concepts and math behind decision trees	✔
Transform the data so that we can use trees	✔
Implement the decision tree algorithm and predict on test	✔
Evaluate the model and store final results	✔
Implement decision tree on the entire dataset and evaluate its results	✔
Optimize the decision tree by tuning the hyperparameters	
Run the optimized model, predict and evaluate the new model	

Ways to optimize a decision tree

- When we originally built our decision tree, we saw that there were many parameters
- All the values of our original tree are set to the `tree` defaults within `sklearn`
- Your manager wants you to optimize the tree **focusing on the four parameters we called out previously**
  - `max_depth = None`
  - `min_samples_split = 2`
  - `min_samples_leaf = 1`
  - `max_features = None`

Define an optimal number function

- Before we optimize individual parameters, let's build a function that will help us store the parameters we will be using in our `optimized_tree`
- The input is:
  - `values` : list of values for the given parameter that we iterate through
  - `test_results` : predictions on the test set for each parameter that we iterate over
- The output is:
  - `best_value` : the actual parameter value that performs best and that we will use in our final optimized tree

```
# Define function that will determine the optimal number for each parameter.
def optimal_parameter(values,test_results):
    best_test_value = max(test_results)
    best_test_index = test_results.index(best_test_value)
    best_value = values[best_test_index]
    return(best_value)
```

Optimize: max depth

- `max_depth` indicates how deep the tree can be
- **The deeper the tree, the more splits it has and captures more information about the data**
- **But remember, there is a fine line between a well fitted model and an *overfitted* model**
- In our original model, `max_depth = None`
- Now, we're going to fit a decision tree with depths ranging from 1 to 32 and plot the training and test accuracy

Optimize: max depth

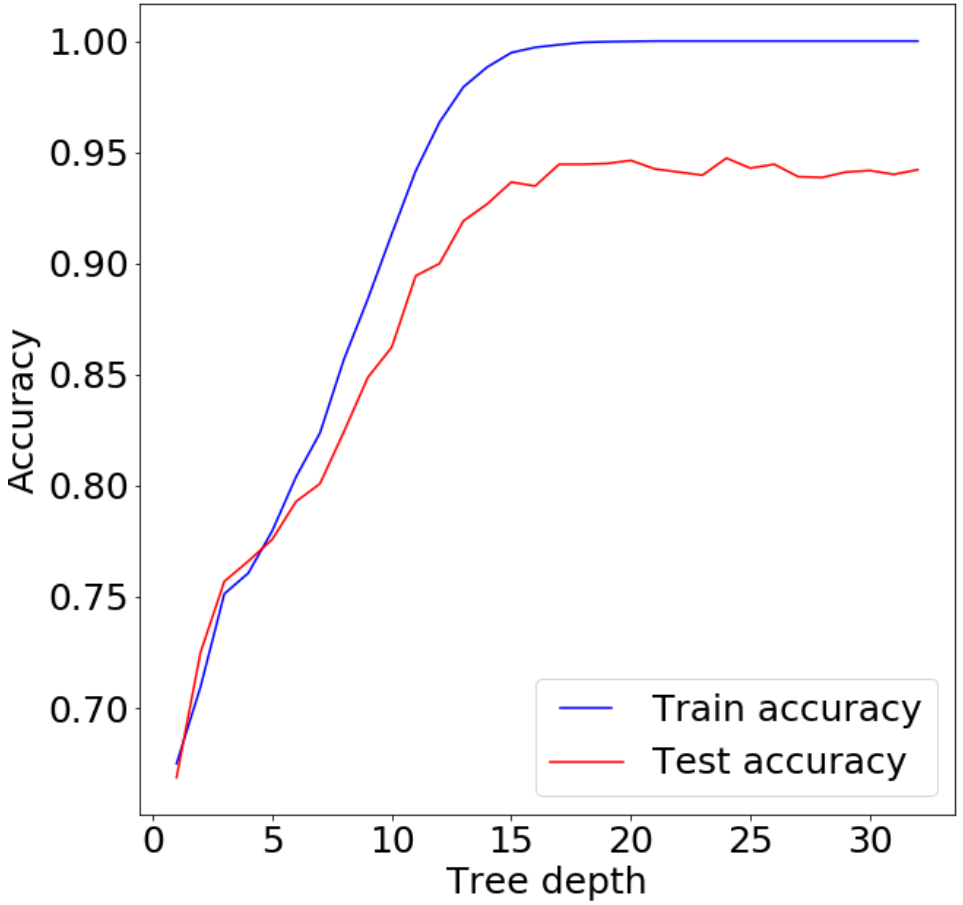
```
# Max depth:
max_depths = np.linspace(1, 32, 32, endpoint = True)
train_results = []
test_results = []
for max_depth in max_depths:
   dt = DecisionTreeClassifier(max_depth = max_depth)
   dt.fit(X_train, y_train)
   train_pred = dt.predict(X_train)
   acc_train = accuracy_score(y_train, train_pred)
   # Add AUC score to previous train results
   train_results.append(acc_train)
   y_pred = dt.predict(X_test)
   acc_test = accuracy_score(y_test, y_pred)
   # Add AUC score to previous test results
   test_results.append(acc_test)

# Store optimal max_depth.
```

```
optimal_max_depth = optimal_parameter(max_depths, test_results)
```

Plot: max depth

- Let's plot the max depth `train_results` and `test_results`
- This will allow us to see when the model starts overfitting, as well as when the optimal test results are achieved
- **What observations can you make?**

```
# Plot max depth over 1 - 32.
line1, = plt.plot(max_depths, train_results,
'b', label = "Train accuracy")
line2, = plt.plot(max_depths, test_results, 'r',
label = "Test accuracy")
plt.legend(handler_map={line1:
HandlerLine2D(numpoints = 2)})
plt.ylabel('Accuracy')
plt.xlabel('Tree depth')
plt.show()
```



Optimize: min samples split

- `min_samples_split` represents the minimum number of samples required to split an internal node
- **This varies between at least one sample at each node to all samples at each node**
- When we **increase this parameter, the tree becomes more constrained** as it has to consider more samples at each node
- We will vary the parameter from 10% to 100% of the samples

Optimize: min samples split

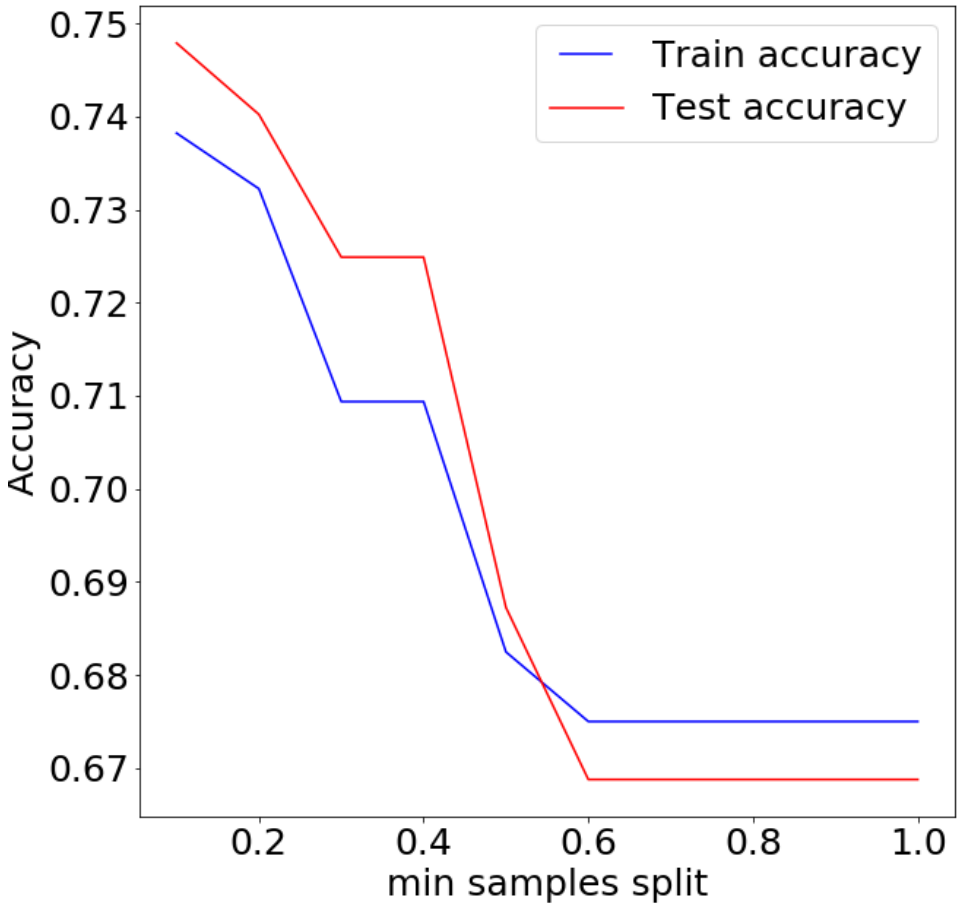
```
min_samples_splits = np.linspace(0.1, 1.0, 10, endpoint = True)
train_results = []
test_results = []
for min_samples_split in min_samples_splits:
   dt = DecisionTreeClassifier(min_samples_split = min_samples_split)
   dt.fit(X_train, y_train)
   train_pred = dt.predict(X_train)
   acc_train = accuracy_score(y_train, train_pred)
   # Add AUC score to previous train results
   train_results.append(acc_train)
   y_pred = dt.predict(X_test)
   acc_test = accuracy_score(y_test, y_pred)
   # Add AUC score to previous test results
   test_results.append(acc_test)

# Store optimal max_depth.
```

```
optimal_min_samples_split = optimal_parameter(min_samples_splits, test_results)
```

Plot: min samples split

- Let's plot the min samples split, `train_results` and `test_results`
- **What observations can you make?**

```
# Plot min_sample split.
line1, = plt.plot(min_samples_splits,
train_results, 'b', label =  "Train accuracy")
line2, = plt.plot(min_samples_splits,
test_results, 'r', label =  "Test accuracy")
plt.legend(handler_map = {line1:
HandlerLine2D(numpoints = 2)})
plt.ylabel('Accuracy')
plt.xlabel('min samples split')
plt.show()
```



Optimize: min samples leaf

- `min_samples_leaf` is the minimum number of samples required to be at a lead node
- This parameter is similar to `min_samples_split` except that **this parameter describes the minimum number of samples at the leafs - the base of the tree**

Optimize: min samples leaf

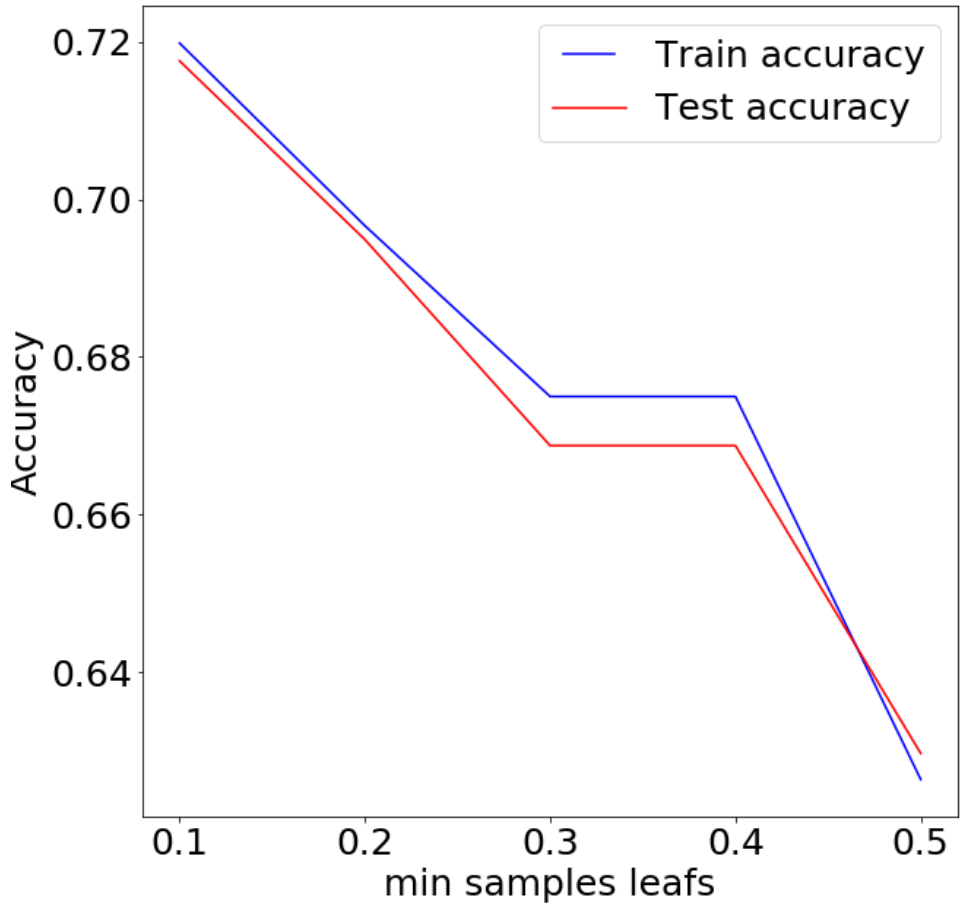
```
# Min_samples_leaf:
min_samples_leafs = np.linspace(0.1, 0.5, 5, endpoint = True)
train_results = []
test_results = []
for min_samples_leaf in min_samples_leafs:
   dt = DecisionTreeClassifier(min_samples_leaf=min_samples_leaf)
   dt.fit(X_train, y_train)
   train_pred = dt.predict(X_train)
   acc_train = accuracy_score(y_train, train_pred)
   # Add AUC score to previous train results
   train_results.append(acc_train)
   y_pred = dt.predict(X_test)
   acc_test = accuracy_score(y_test, y_pred)
   # Add AUC score to previous test results
   test_results.append(acc_test)
```

```
optimal_min_samples_leafs = optimal_parameter(min_samples_leafs, test_results)
```

Plot: min samples leaf

- Let's plot the min samples leaf `train_results` and `test_results`
- **What observations can you make?**

```
# Plot min_sample split.
line1, = plt.plot(min_samples_leafs,
train_results, 'b', label =  "Train accuracy")
line2, = plt.plot(min_samples_leafs,
test_results, 'r', label =  "Test accuracy")
plt.legend(handler_map = {line1:
HandlerLine2D(numpoints = 2)})
plt.ylabel('Accuracy')
plt.xlabel('min samples leafs')
plt.show()
```



Optimize: max features

- `max_features` represents the number of features to consider when looking for the best split
- This parameter is set to `None` as its default value, meaning the tree will always look through all features
- This could sometimes cause overfitting and / or is computationally expensive when working with many variables

Optimize: max features

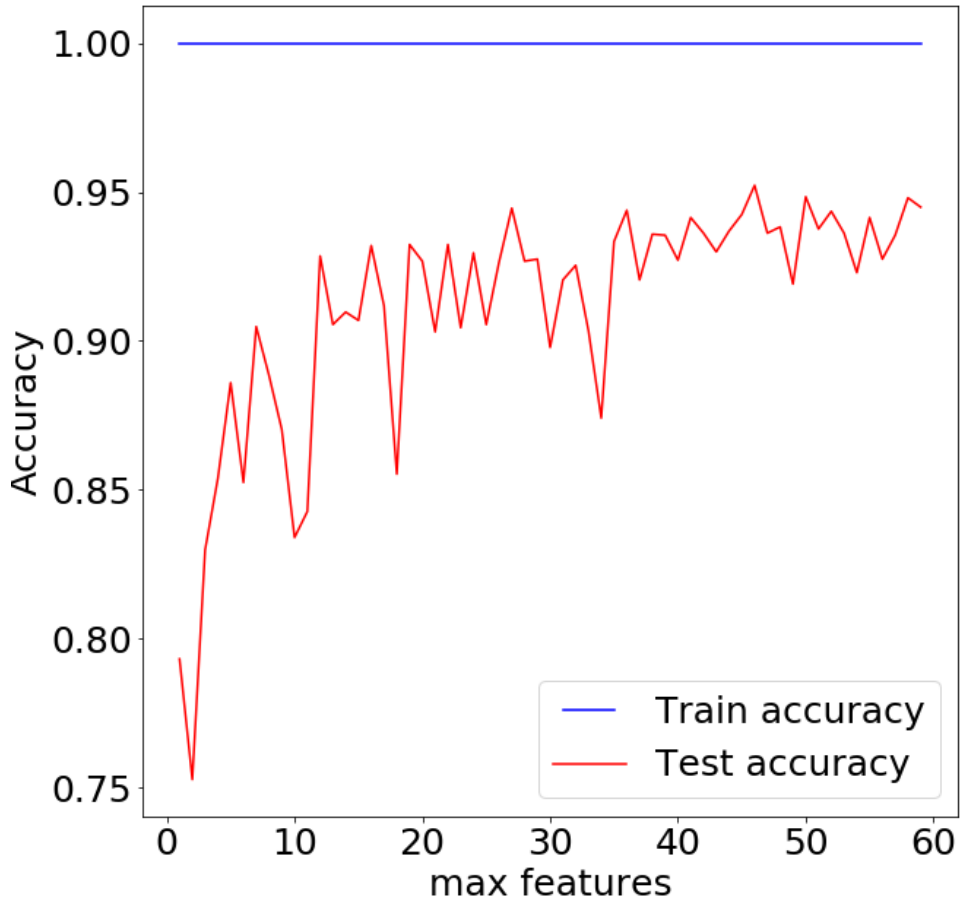
```
# Max_features:
max_features = list(range(1, X.shape[1]))
train_results = []
test_results = []
for max_feature in max_features:
   dt = DecisionTreeClassifier(max_features = max_feature)
   dt.fit(X_train, y_train)
   train_pred = dt.predict(X_train)
   acc_train = accuracy_score(y_train, train_pred)
   # Add AUC score to previous train results
   train_results.append(acc_train)
   y_pred = dt.predict(X_test)
   acc_test = accuracy_score(y_test, y_pred)
   # Add AUC score to previous test results
   test_results.append(acc_test)
```

```
optimal_max_features = optimal_parameter(max_features, test_results)
```

Plot: max features

- Let's plot the max features, `train_results` and `test_results`
- **What observations can you make?**

```
# Plot min_sample split.
line1, = plt.plot(max_features, train_results,
'b', label = "Train accuracy")
line2, = plt.plot(max_features, test_results,
'r', label = "Test accuracy")
plt.legend(handler_map = {line1:
HandlerLine2D(numpoints = 2)})
plt.ylabel('Accuracy')
plt.xlabel('max features')
plt.show()
```



Module completion checklist

Objective	Complete
Discuss use cases for decision trees and random forests	✔
Summarize the concepts and math behind decision trees	✔
Transform the data so that we can use trees	✔
Implement the decision tree algorithm and predict on test	✔
Evaluate the model and store final results	✔
Implement decision tree on the entire dataset and evaluate its results	✔
Optimize the decision tree by tuning the hyperparameters	✔
Run the optimized model, predict and evaluate the new model	

Optimized model

- We have now walked through four parameters that will help us optimize our decision tree
- Remember that when we optimized each parameter, we saved the optimal parameter using our `optimal_parameter` function
- Let's look at what each of the optimal parameters are

```
print("The optimal max depth is:",
optimal_max_depth)
```

```
The optimal max depth is: 24.0
```

```
print("The optimal min samples split is:",
optimal_min_samples_split)
```

```
The optimal min samples split is: 0.1
```

```
print("The optimal min samples leaf is:",
optimal_min_samples_leafs)
```

```
The optimal min samples leaf is: 0.1
```

```
print("The optimal max features is:",
optimal_max_features)
```

```
The optimal max features is: 46
```

Build optimized model

- Now, we will run the optimized model on our `X_train`

```
# Set the seed.
np.random.seed(1)

# Implement the decision tree on X_train.
clf_optimized = tree.DecisionTreeClassifier(max_depth = optimal_max_depth,
                                            min_samples_split = optimal_min_samples_split,
                                            min_samples_leaf = optimal_min_samples_leafs,
                                            max_features = optimal_max_features)

# We can now see our optimized features where before they were just default:
print(clf_optimized)
```

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=24.0,
                       max_features=46, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=0.1, min_samples_split=0.1,
                       min_weight_fraction_leaf=0.0, presort=False,
                       random_state=None, splitter='best')
```

```
clf_optimized_fit = clf_optimized.fit(X_train, y_train)
```

Predict with optimized model

- Finally, let's predict on `X_test` and calculate our accuracy score
- **How is our optimized model doing?**
- **What other metrics can you also look at?**

```
# Predict on X_test.
y_predict_optimized = clf_optimized_fit.predict(X_test)

# Get the accuracy score.
acc_score_tree_optimized = accuracy_score(y_test, y_predict_optimized)

print(acc_score_tree_optimized)
```

```
0.7182705718270572
```

Predict and save results

- Now we know some of the parameters that help us optimize our decision tree
- **What is another way you could optimize the tree, instead of searching each parameter separately?**
- *Hint: We used it for kNN*

Predict and save results

- This other method is `GridSearchCV` and is helpful but can be computationally expensive
- Going through each of the four parameters helps you understand your tree a little better
- **Let's save our optimized tree accuracy score in our `model_final` dataset**
- **And finally, pickle our `model_final` dataset with all three tree model accuracy scores saved**

```
# Add the optimized model to our dataframe.
model_final_tree = model_final_tree.append(
            {'metrics' : "accuracy" ,
             'values' : round(acc_score_tree_optimized,4),
             'model':'tree_all_variables_optimized' } ,
              ignore_index = True)
print(model_final_tree)
```

```
    metrics  values                          model
0  accuracy  0.6046                          knn_5
1  accuracy  0.6188                knn_GridSearchCV
2  accuracy  0.6287                         knn_29
3  accuracy  0.6356                       logistic
4  accuracy  0.7845          logistic_whole_dataset
5  accuracy  0.7859                  logistic_tuned
6  accuracy  0.6611              tree_simple_subset
7  accuracy  0.9407              tree_all_variables
8  accuracy  0.7183  tree_all_variables_optimized
```

```
pickle.dump(model_final_tree,
open("model_final_tree_all.sav","wb" ))
```

Next: ensemble methods

- We have now learned about classification models like:
  - kNN
  - logistic regression
- As well as a machine learning method:
  - decision trees

Next up, we will be learning about more powerful machine learning algorithms, ensemble methods

Get excited!

Knowledge check 4

Exercise 3

Module completion checklist

Objective	Complete
Discuss use cases for decision trees and random forests	✔
Summarize the concepts and math behind decision trees	✔
Transform the data so that we can use trees	✔
Implement the decision tree algorithm and predict on test	✔
Evaluate the model and store final results	✔
Implement decision tree on the entire dataset and evaluate its results	✔
Optimize the decision tree by tuning the hyperparameters	✔
Run the optimized model, predict and evaluate the new model	✔

Workshop: Next steps!

- Workshops are to be completed in the afternoon either with a dataset for a capstone project or with another dataset of your choosing
- Make sure to annotate and comment your code
- This is an exploratory exercise to get you comfortable with the content we discussed today

Today you will

- Run a decision model
- Use performance metrics covered in class to assess the model
- Evaluate the optimal hyperparameters using GridSearchCV and build the optimal model
- Save accuracy metrics of all the models to the `model_final_workshop` dataframe

This completes our module

Congratulations!