DATA SOCIETY®

Advanced classification - day 1

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

Module completion checklist

Objective	Complete
Introduce random forest and discuss use cases	
Summarize the concepts associated with random forest and bagging	
Load transformed dataset and implement random forest	
Introduce gradient boosting and how it compares to bagging	
Discuss gradient tree boosting within scikit-learn and implement on Costa Rican data	

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"
```

Loading packages

- Let's load the packages we will be using
- These packages are used for classification using decision trees, random forests, xgboost and other tools

```
import os
import pickle
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from textwrap import wrap
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn import tree
from sklearn.metrics import accuracy_score

# New today - random forest and boosting packages
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from matplotlib.legend_handler import HandlerLine2D
```

Working directory

- Set working directory to `data_dir`

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

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

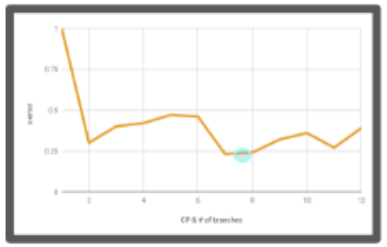
```
/home/[user-name]/Desktop/af-werx/data
```

Recap: decision trees process

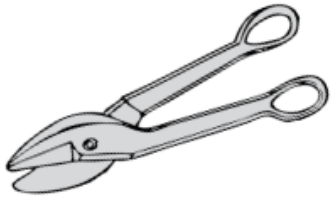
- We will be reviewing decision trees as they are related to random forests

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 -			

Recap: decision trees

Decision trees are **great** when used for:

- Classification and regression
- Handling numerical and categorical data
- Handling data with missing values
- Handling data with nonlinear 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

Random forest

- We understand how one **tree** works
- Why not try a **forest** ?

What is Random forest?

- Ensemble method used for **classification and regression tasks**
- Supervised learning algorithm which builds **multiple decision trees** and aggregates the result
- Uses a technique called Bootstrap Aggregation, commonly known as **Bagging**
- Limits overfitting and bias error

Why should we use random forest?

- Reduction in overfitting
- Higher predictive accuracy
- Efficient with large datasets
- When should we use **decision trees** instead?
  - Intuitive and easily interpretable results
  - Less computationally expensive algorithm

Random forest: use cases

- The random forest algorithm is used in a **multitude of industries** such as banking, medicine, e-commerce, etc.
- Some examples are:
  - Fraud detection
  - Identifying a disease by analyzing patient's medical history
  - Predicting the behavior of the stock market
  - Understanding whether a customer will buy a product or not

Goal for today

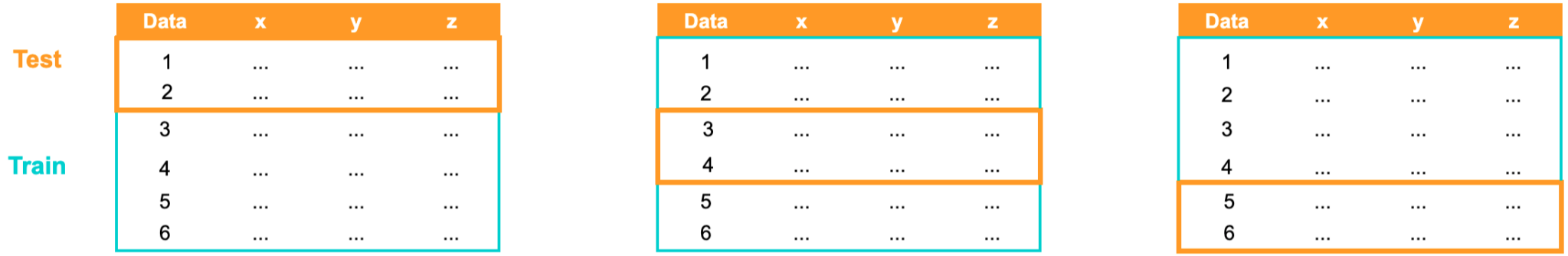
- Compare a **random forest** and **gradient boosting** against each other to predict poverty levels in the Costa Rican dataset
- Implement both models and save results
- Compare models at the end of today and decide on your **model champion**

Module completion checklist

Objective	Complete
Introduce random forest and discuss use cases	✔
Summarize the concepts associated with random forest and bagging	
Load transformed dataset and implement random forest	
Introduce gradient boosting and how it compares to bagging	
Discuss gradient tree boosting within scikit-learn and implement on Costa Rican data	

Why is random forest popular?

- It uses many decision trees on different subsections of the dataset and averages out the results to improve the predictive accuracy and control overfitting
- **"Bagging"** is an ensemble method that adopts the bootstrap sampling technique, which creates new datasets by using random sampling with replacement
- The **O**ut **o**f **B**ag error rate for the forest of trees is used as a metric to assess the algorithms' performance
- They use a built-in form of multi-fold cross-validation method



Bagging observations

- **Bootstrap aggregation** is the process that makes up bagging
- **Bootstrap sampling technique** creates new datasets by random sampling with replacement
- **Bagging** within **CART** lets you choose how many trees, i.e., how many bootstrapped sampled training sets to create

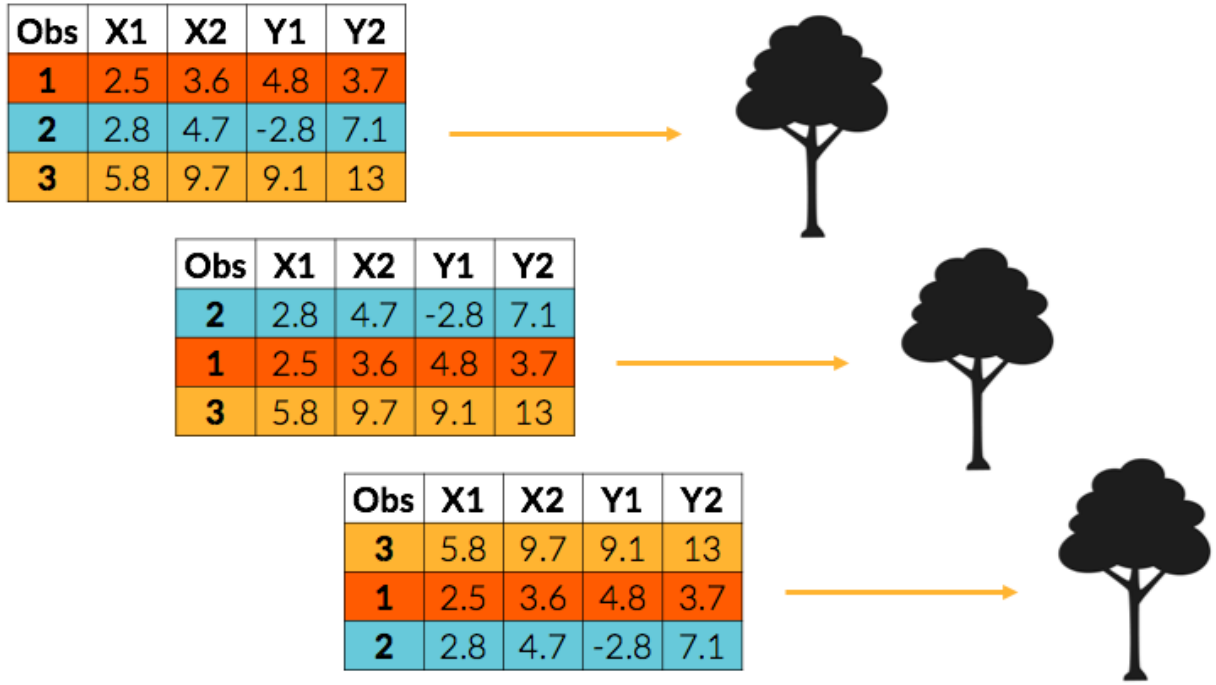
Random forest: bagging

Bagging of observations in itself is pretty cool!

But that's not all it does...

Bagging is not enough

- With using just bagging, random forests can have a lot of structural similarities with decision trees and, in turn, have a high **bias** (i.e. a known drawback of tree algorithms!)



Sample predictors as well!

- The true power of **random forests** vs **CART** is the limitation of predictors
- For each tree, both samples of observations and **random samples of features** are used instead of using the entire set of features every time

Obs	X1	X2
1	2.5	3.6
1	2.5	3.6
3	5.8	9.7

Obs	X1	Y1
2	2.8	-2.8
1	2.5	4.8
2	2.8	-2.8

Obs	X2	Y1
3	9.7	9.1
3	9.7	9.1
2	4.7	-2.8

- The resulting model becomes **unbiased** due to a good tree variety, where no variable dominates!

Building the forest

The two main parameters we need to set to build a **random forest** are:

1. Number of trees
2. Number of features per tree

We can stick with these rules:

1. **N of trees** - the more the better, but a good rule of thumb is $n \approx 100$, where $n = number\ of\ trees$
2. **N of features per tree** - the rule of thumb here is $m = \sqrt{p}$, where $p = number\ of\ predictors$

Random forest methodology

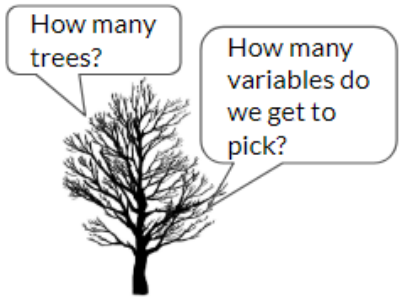
Step 1:
Set *n* & *m* for forest

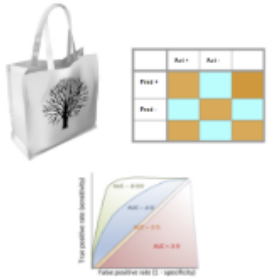
Step 2:
Build forest on training data

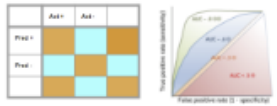
Step 3:
Check performance

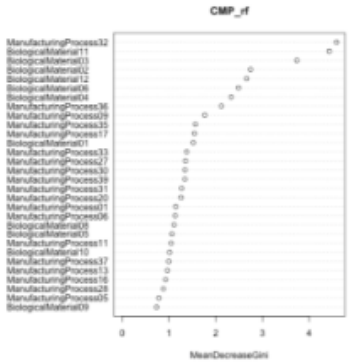
Step 4:
Apply to test data & evaluate

Step 5:
Use variable importance as needed

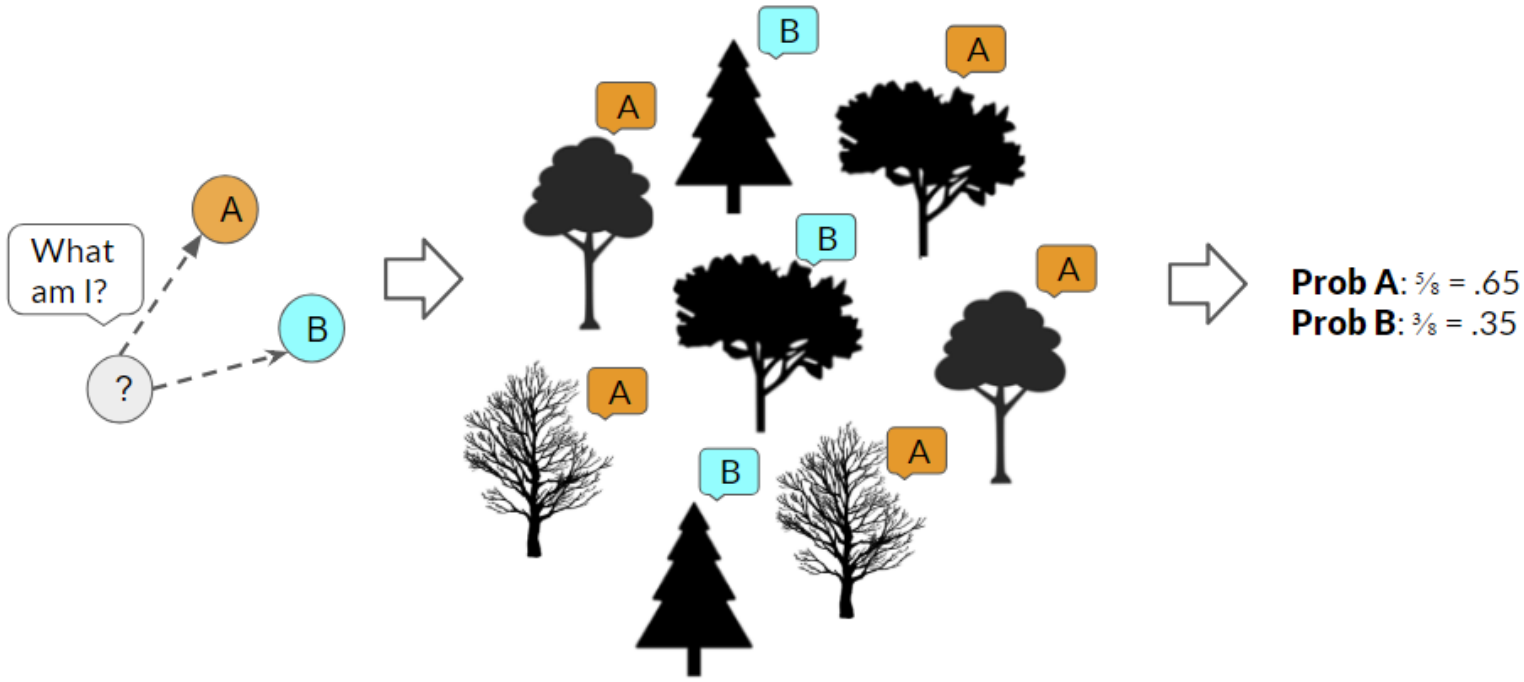




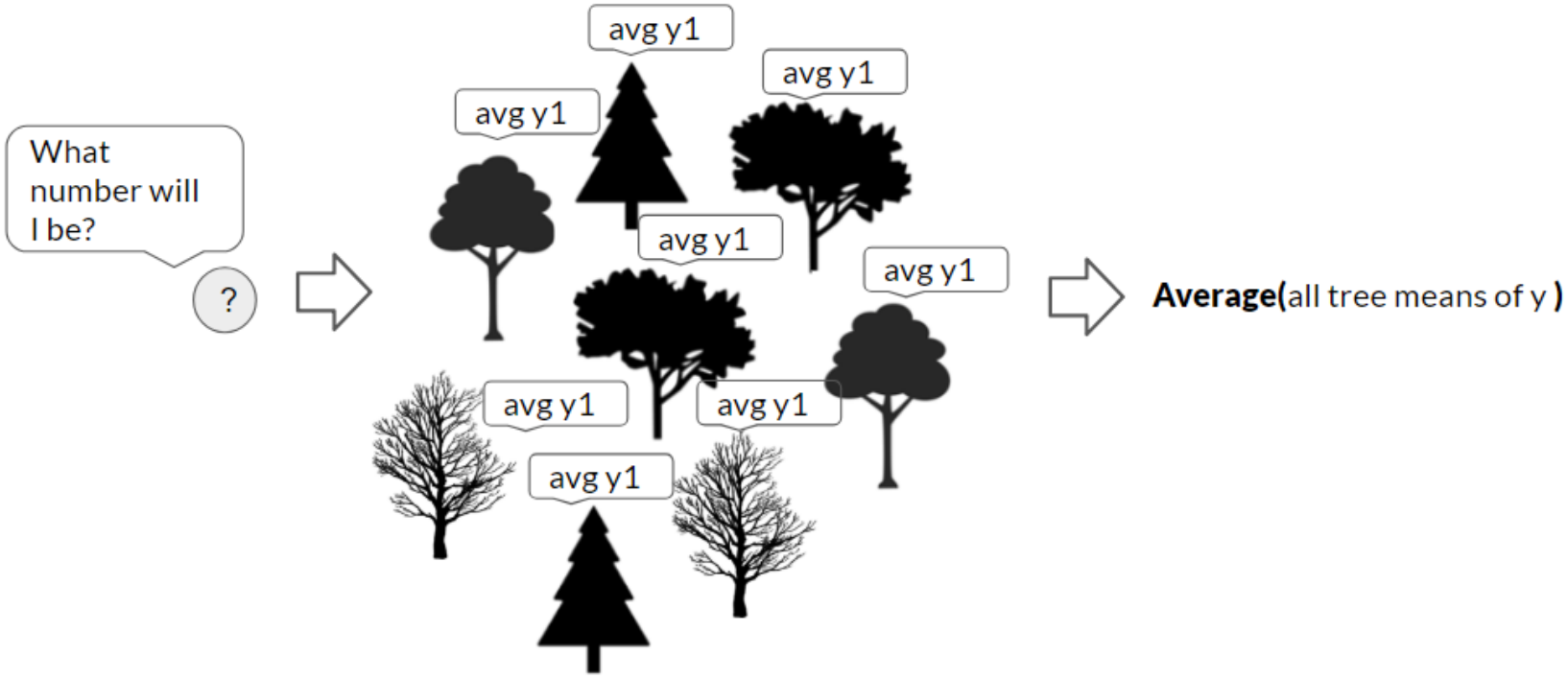




Random forest classification



Handling regression with random forest



For this module, we will focus on classification

Knowledge Check 1

Module completion checklist

Objective	Complete
Introduce random forest and discuss use cases	✔
Summarize the concepts associated with random forest and bagging	✔
Load transformed dataset and implement random forest	
Introduce gradient boosting and how it compares to bagging	
Discuss gradient tree boosting within scikit-learn and implement on Costa Rican data	

Datasets for today

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

- **A dataset in class to learn the concepts**
  - Costa Rican dataset

- **A dataset for our in-class exercises**
  - Chicago census dataset

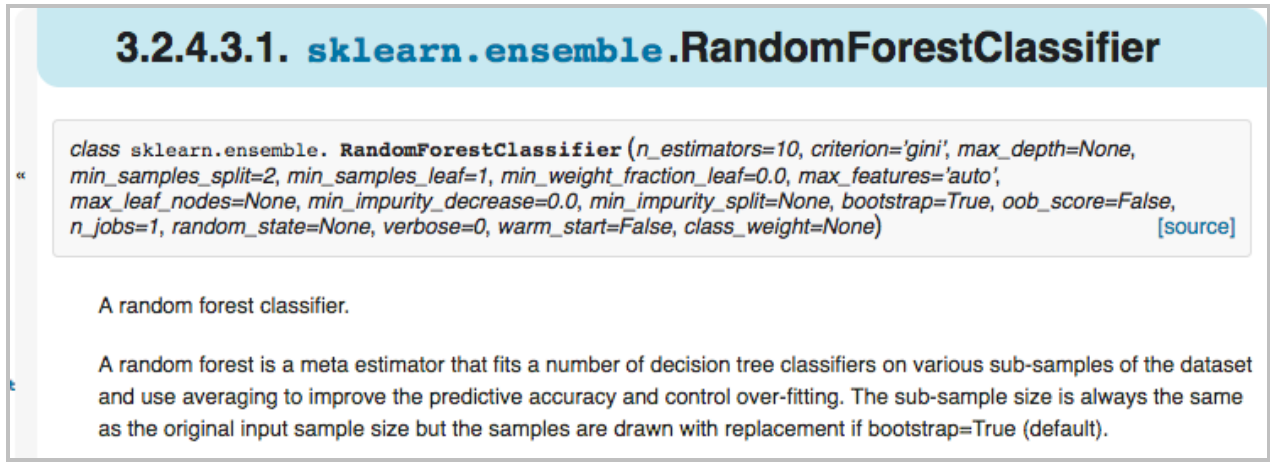
Scikit learn

- We will be using the well known Python library `scikit-learn` today
- Scikit-learn is used for many machine learning algorithms
- Here is a quick overview of some of the popular methods scikit-learn touches

ML methods	Complete
Clustering	Unsupervised learning methods such as k-means
Classification and regression	Supervised learning methods like generalized linear models, logistic regression, support vector machines, and decision trees
Cross validation	Estimating the performance of supervised models
Dimensionality reduction	Feature selection and feature extraction methods
Ensemble methods	Combining predictions of multiple supervised models
Parameter tuning	Adjusting model parameters to get the most out of models
Manifold learning	Summarizing and depicting complex multi-dimensional data

scikit-learn: random forest

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



3.2.4.3.1. sklearn.ensemble.RandomForestClassifier

class sklearn.ensemble. **RandomForestClassifier** (*n_estimators=10, criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=1, random_state=None, verbose=0, warm_start=False, class_weight=None*) [source]

A random forest classifier.

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is always the same as the original input sample size but the samples are drawn with replacement if bootstrap=True (default).

- The sub-sample size is always the same as the original input sample size but the samples are drawn with replacement as long as `bootstrap = True` (default)
- For all the parameters of the `tree` package, visit ***scikit-learn's documentation***

Review data cleaning steps

- Last week, we moved from using a few variables from the `costa_rica_poverty` dataset **to all the variables**
- When we did this, we encountered **highly correlated variables** within the dataset
- **Today, we will be loading the cleaned dataset from last week, that we pickled**
- To recap, the steps to get to this cleaned dataset were:
  - Remove household ID and individual ID
  - Remove variables with over 50% NAs
  - Transform target variable to binary
  - Remove highly correlated variables
- Your manager would like for you to implement a random forest on the Costa Rica dataset in order to improve our accuracy with prediction

Load the cleaned dataset

- Let's load the dataset from last week, `costa_no_hc` - no highly correlated variables
- Save it as `costa_clean`

```
os.chdir(data_dir)
```

```
costa_clean = pickle.load(open("costa_no_hc.sav","rb"))
```

```
print(costa_clean.head())
```

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

[5 rows x 61 columns]
```

Print info on data

- Let's view the column names

```
costa_clean.columns
```

```
Index(['rooms', 'tablet', 'males_under_12', 'males_over_12',
       'females_under_12', 'females_over_12', 'years_of_schooling',
       'wall_block_brick', 'wall_socket', 'wall_prefab_cement', 'wall_wood',
       'floor_mos_cer_terr', 'floor_wood', 'ceiling', 'electric_public',
       'toilet_sewer', 'cookenergy_elec', 'trash_truck', 'wall_bad',
       'wall_reg', 'roof_bad', 'roof_reg', 'floor_bad', 'floor_reg',
       'disabled_ppl', 'male', 'under10', 'free', 'married', 'separated',
       'single', 'hh_head', 'hh_spouse', 'hh_child', 'num_65plus',
       'dependency_rate', 'male_hh_head_educ', 'female_hh_head_educ',
       'meaneduc', 'educ_primary_inc', 'educ_primary', 'educ_secondary_inc',
       'educ_secondary', 'educ_undergrad', 'ppl_per_room', 'house_owned_full',
       'house_owned_paying', 'house_rented', 'house_other', 'computer',
       'television', 'num_mobilephones', 'region_central', 'region_Chorotega',
       'region_pacifico', 'region_brunca', 'region_antlantica',
       'region_huetar', 'urban_zone', 'age', 'Target'],
      dtype='object')
```

Split into training and test sets

```
# Select the predictors and target.
X = costa_clean.drop(['Target'], axis = 1)
y = np.array(costa_clean['Target'])

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

# Split into the training and test sets.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3)
```

RandomForestClassifier

- We introduced the package `RandomForestClassifier` earlier today
- We are now going to use it to build a random forest on our clean data
- First, let's look at the **methods** available once the model is built

Methods

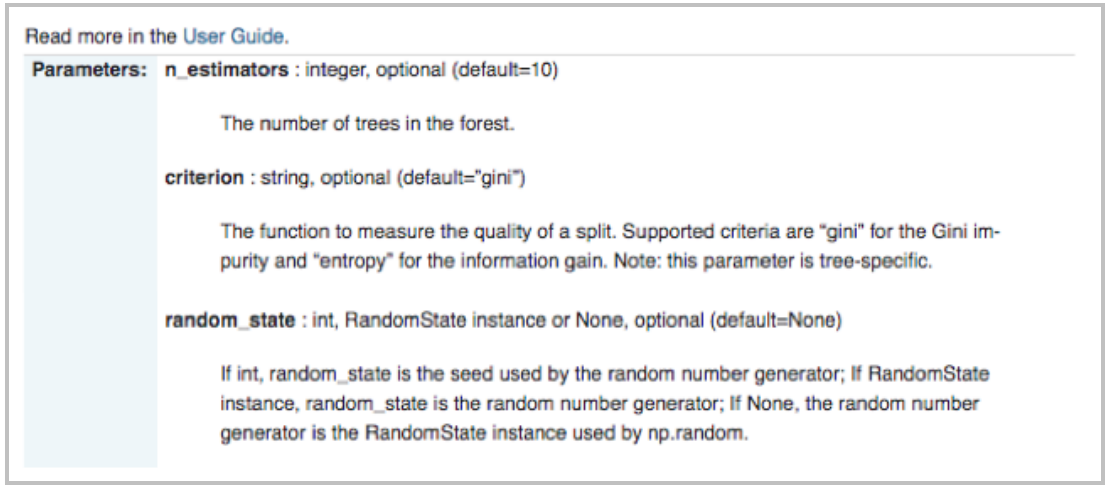
`apply` (X)	Apply trees in the forest to X, return leaf indices.
`decision_path` (X)	Return the decision path in the forest
`fit` (X, y[, sample_weight])	Build a forest of trees from the training set (X, y).
`get_params` ([deep])	Get parameters for this estimator.
`predict` (X)	Predict class for X.
`predict_log_proba` (X)	Predict class log-probabilities for X.
`predict_proba` (X)	Predict class probabilities for X.
`score` (X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params` (**params)	Set the parameters of this estimator.

- We are going to:
  - **Build** the random forest model
  - **Fit** the model to the training data
  - **Predict** on the test data using our trained model

Building our model

- Let's build our random forest model and use all default parameters for now, as our baseline model

```
forest = RandomForestClassifier(criterion = 'gini', n_estimators = 100,
random_state = 1)
```



Read more in the User Guide.

Parameters: **n_estimators** : integer, optional (default=10)

The number of trees in the forest.

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. Note: this parameter is tree-specific.

random_state : int, RandomState instance or None, optional (default=None)

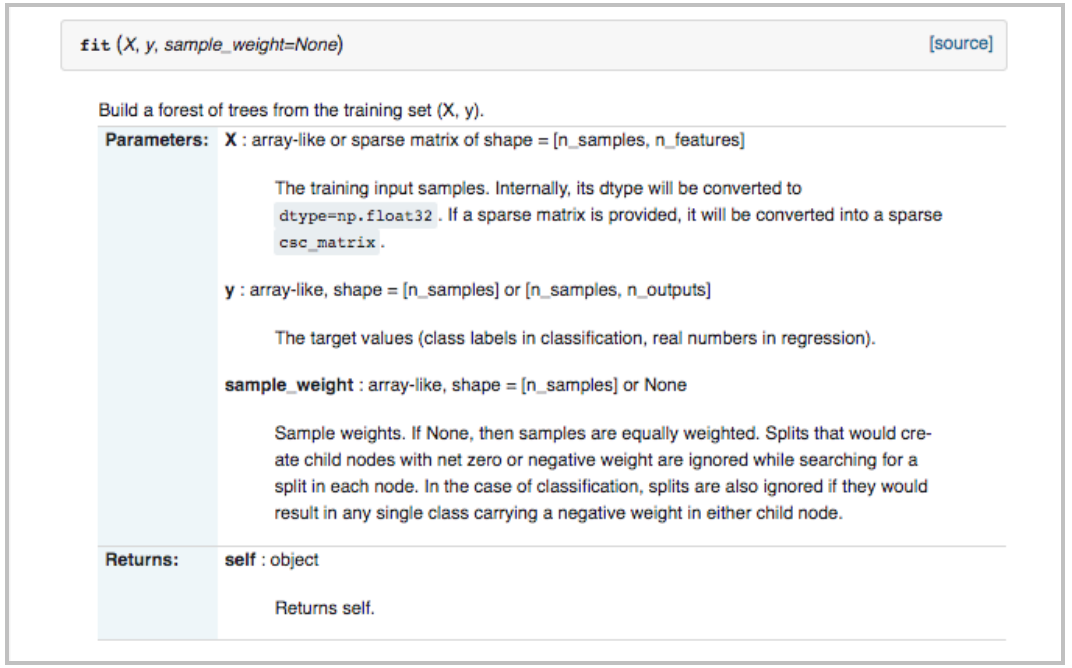
If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random.

Fitting our model

- **Fit** the model to the training data

```
# Fit the saved model to your training data.
forest.fit(X_train, y_train)
```

```
RandomForestClassifier(bootstrap=True,
class_weight=None, criterion='gini',
                       max_depth=None,
max_features='auto', 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, n_estimators=100,
                       n_jobs=None,
oob_score=False, random_state=1, verbose=0,
                       warm_start=False)
```



`fit` (*X, y, sample_weight=None*) [source]

Build a forest of trees from the training set (X, y).

Parameters:

X : array-like or sparse matrix of shape = [n_samples, n_features]

The training input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`.

y : array-like, shape = [n_samples] or [n_samples, n_outputs]

The target values (class labels in classification, real numbers in regression).

sample_weight : array-like, shape = [n_samples] or None

Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. In the case of classification, splits are also ignored if they would result in any single class carrying a negative weight in either child node.

Returns:

self : object

Returns self.

Predicting with our data

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

```
# Predict on test data.
y_predict_forest = forest.predict(X_test)

# Look at the first few predictions.
print(y_predict_forest[0:5,])
```

```
[False False False False False]
```

Confusion matrix and accuracy

Both `confusion_matrix` and `accuracy_score` take 2 arguments:

1. Original data labels
2. Predicted labels

```
# Take a look at test data confusion matrix.
conf_matrix_forest = metrics.confusion_matrix(y_test, y_predict_forest)
print(conf_matrix_forest)
```

```
[[1760   46]
 [ 144  918]]
```

```
accuracy_forest = metrics.accuracy_score(y_test, y_predict_forest)
print("Accuracy for random forest on test data: ", accuracy_forest)
```

```
Accuracy for random forest on test data:  0.9337517433751743
```

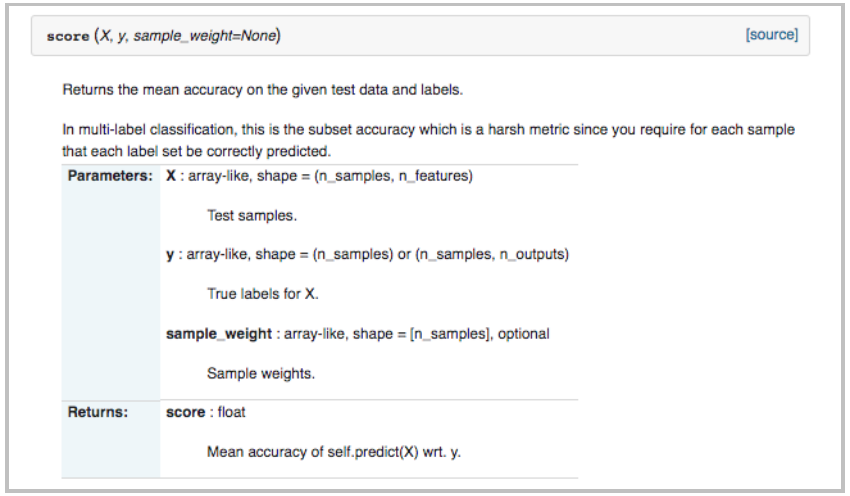
Accuracy of the training dataset

- Let's look at the accuracy of the model we just built, on the training data

```
# Compute accuracy using training data.
acc_train_forest = forest.score(X_train, y_train)

print ("Train Accuracy:", acc_train_forest)
```

```
Train Accuracy: 1.0
```

- 1.0 is high for accuracy
- Remember, this is accuracy on the **training** dataset
- This will not be the same result that you will see on the **test** dataset



score (*X, y, sample_weight=None*) [source]

Returns the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters:

X : array-like, shape = (n_samples, n_features)

Test samples.

y : array-like, shape = (n_samples) or (n_samples, n_outputs)

True labels for X.

sample_weight : array-like, shape = [n_samples], optional

Sample weights.

Returns:

score : float

Mean accuracy of self.predict(X) wrt. y.

Save final accuracy

- **Let's save our random forest score in our `model_final` dataset**
- We first have to load our `model_final` dataframe from last week

```
model_final_forest_gbm =
pickle.load(open("model_final_tree_all.sav","rb"))
```

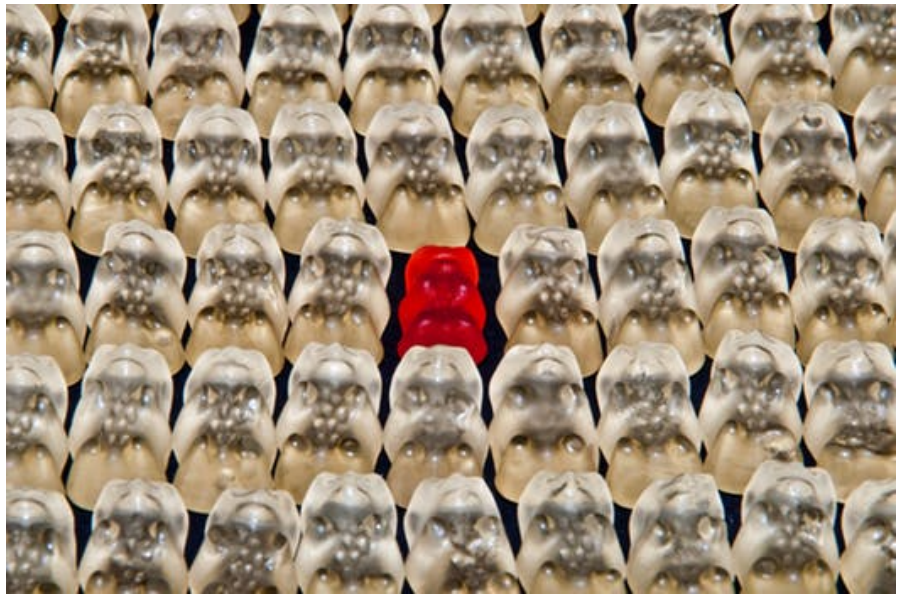
Save final accuracy

```
# Add the model to our dataframe.
model_final_forest_gbm = model_final_forest_gbm.append({'metrics' : "accuracy" ,
                                                        'values' : round(accuracy_forest,4),
                                                        'model':'random forest' } ,
                                                        ignore_index = True)
print(model_final_forest_gbm)
```

```
    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
9  accuracy  0.9338                 random forest
```

Utilizing feature importance

- One benefit of random forests and gradient boosting is that we can look at **feature importance**
- Often times the audience is interested in the outcome, but may not want to understand the details of the algorithm
- Therefore, **we can benefit from illustrating findings through visualizations**
- A **feature importance plot** will show the importance of the feature based on the decrease / increase in rate of error or gain in impurity measure that is present when the feature is present



Applying feature importance on our data

- Your manager would like for you to focus more on certain features within our data
- You should find the importance of the feature accordingly by rate of error or gain in impurity measure
- This will allow for a clearer understanding of our algorithm to the manager and potentially a larger audience

Subsetting our features

- Let's subset the features into another variable named `costarica_features`

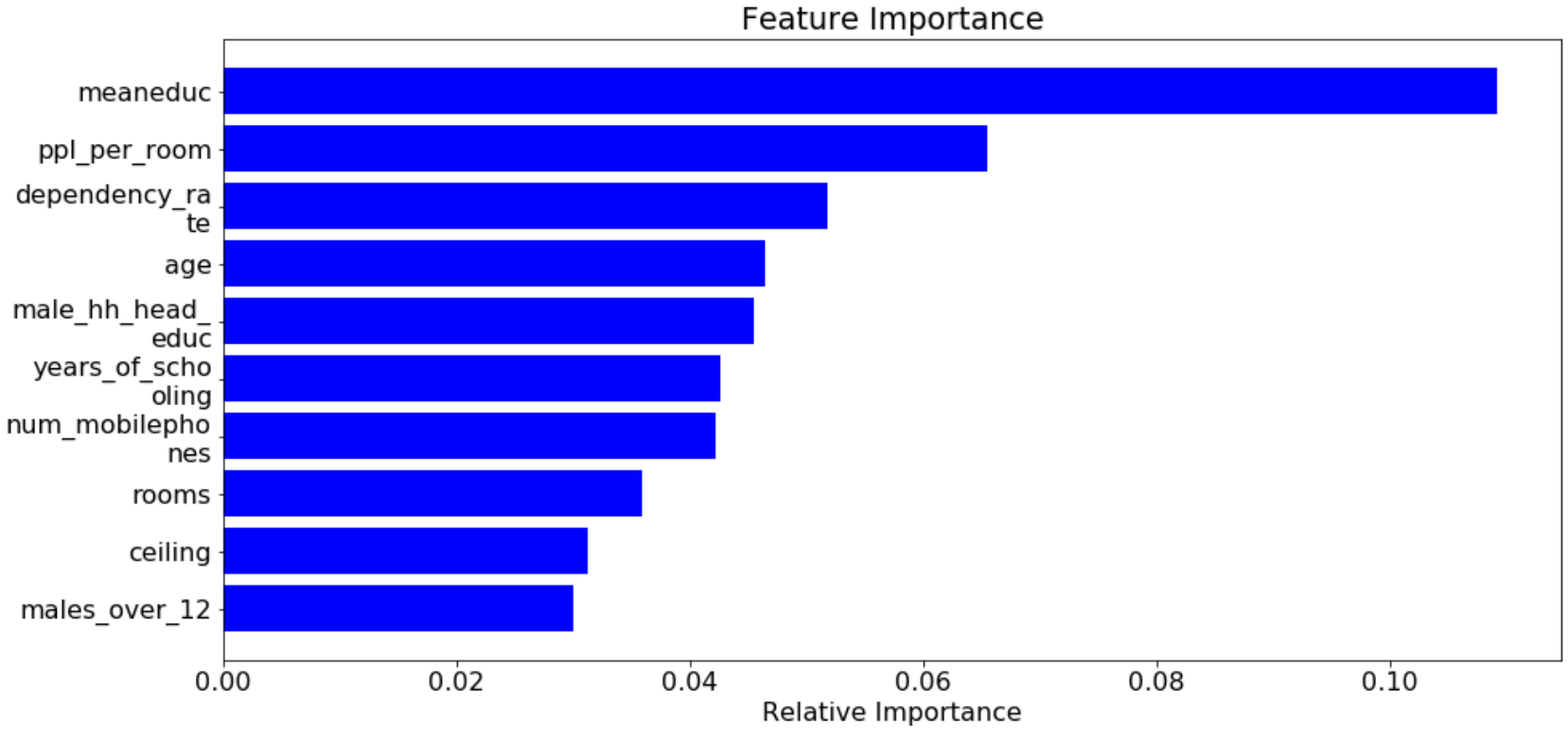
```
costarica_features = costa_clean.drop('Target', axis = 1)
```

- Here is our feature importance plot for the random forest we just built

```
features = costarica_features.columns
importances = forest.feature_importances_
indices = np. argsort(importances)[::-1]
top_indices = indices[0:10][::-1]

plt.figure(1)
plt.title('Feature Importance')
plt.barh(range(len(top_indices)), importances[top_indices], color = 'b', align = 'center')
labels = features[top_indices]
labels = [ '\n'.join(wrap(l,13)) for l in labels ]
plt.yticks(range(len(top_indices)), labels)
plt.xlabel('Relative Importance')
```

- We will compare it to gradient boosting at the end of this module

Feature importance plot



Save the random forest model

- We are going to `pickle` our model now so that we can use it in the next class
- **We will be optimizing the model in the next class!**

```
pickle.dump(forest, open("model_forest.sav","wb" ))
```

Knowledge Check 2

Exercise 1

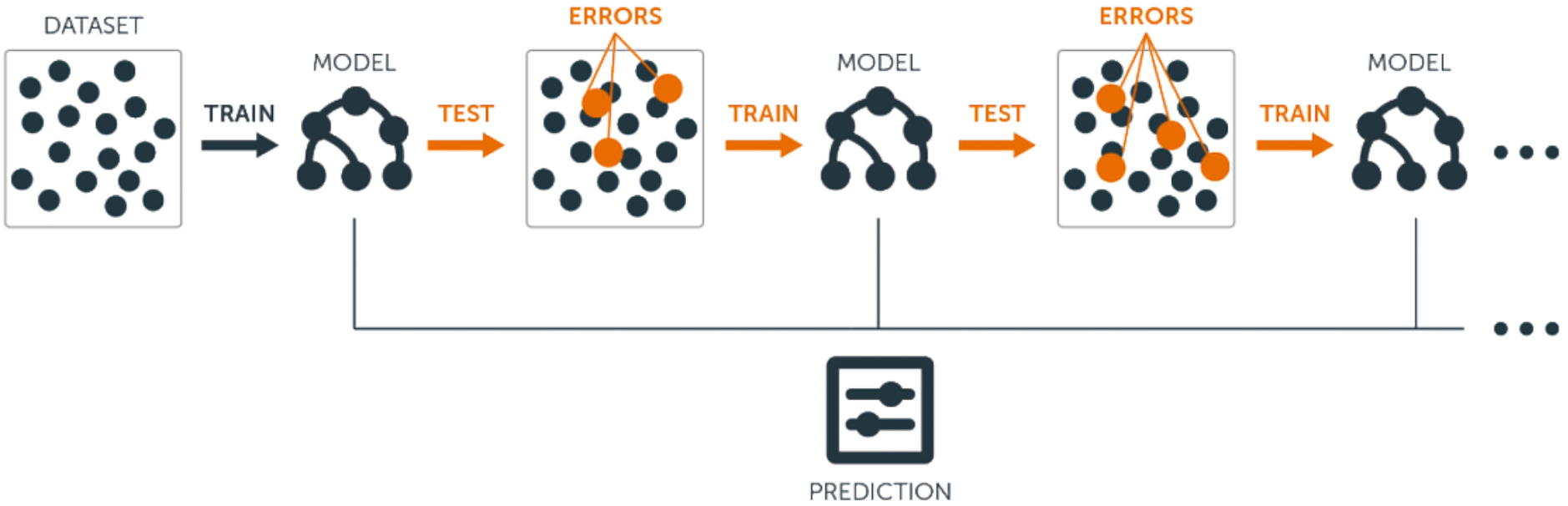
Module completion checklist

Objective	Complete
Introduce random forest and discuss use cases	✔
Summarize the concepts associated with random forest and bagging	✔
Load transformed dataset and implement random forest	✔
Introduce gradient boosting and how it compares to bagging	
Discuss gradient tree boosting within scikit-learn and implement on Costa Rican data	

Gradient boosted trees

- We're now going to learn about another **ensemble method** called **boosting**
- It is an ensemble method because it is a combination of many models on the same dataset
- The primary focus of boosting is to combine **many weak learners** into **one strong learner**
- New predictors are made from the mistakes of the previous predictors

Gradient boosted trees



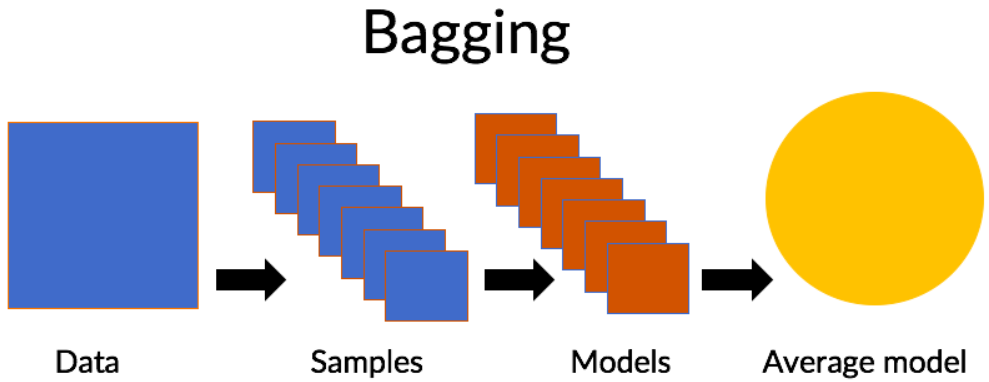
Source

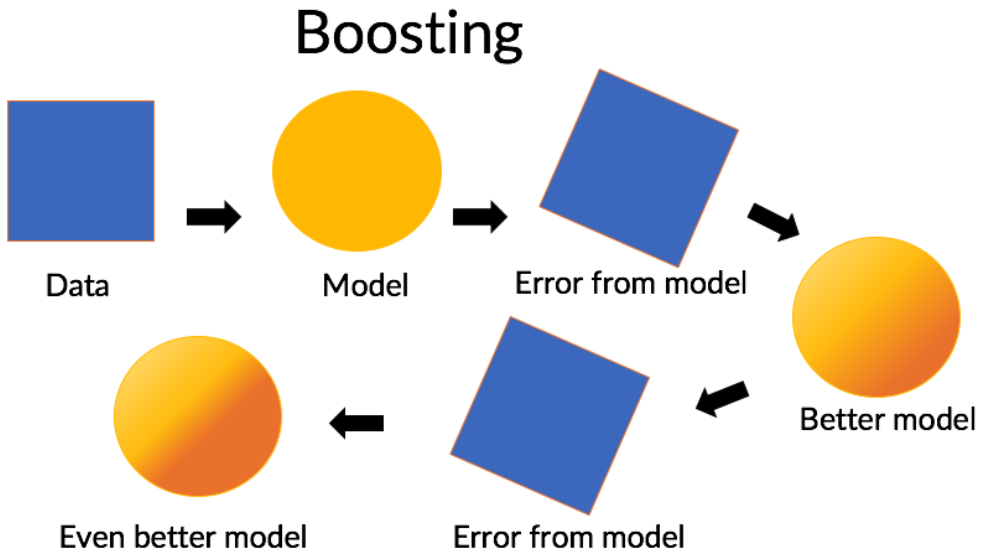
Boosting vs bagging

- We reviewed the concept of **bagging** earlier today with random forests
- Let's see how **boosting** compares to **bagging**

	Bagging	Boosting
Partitioning of the data into subsets	Random	Gives misclassified samples higher preference
Goal	Minimize variance	Increase predictive force
Function to combine single models	Weighted average	Weighted majority vote

Boosting vs bagging





- Unlike bagging, predictors are not made independently, but **sequentially**

Gradient boosting applied to decision trees

- In simple linear regression, you can clearly see the residuals, which are the multiple points around the linear model
- Let's think of these residuals, but apply the concept to decision trees
- When **gradient boosting** uses decision trees, it follows these three steps:
  - Sees the **errors from a decision tree** on the dataset
  - **Identifies the pattern of the errors and builds a new decision tree** on them
  - **Repetitively leverages these patterns in residuals to strengthen the overall model**

Gradient boosting process

- The process of **gradient boosting** is math heavy and complex
- However, for now, it can be simplified to three steps that we just discussed:

1. Fit a decision tree model to the data
2. Fit a decision tree model to the residuals
3. Create a new model

- **Gradient boosting** can be used with classification or regression
- The generalization of the multiple weak learners occurs by the optimization of a differentiable **loss function**
- The **loss function** will change based on the model's target variable:
  - **Regression**: ***gradient descent*** used to minimize mean squared error
  - **Binary classification**: ***logistic function***

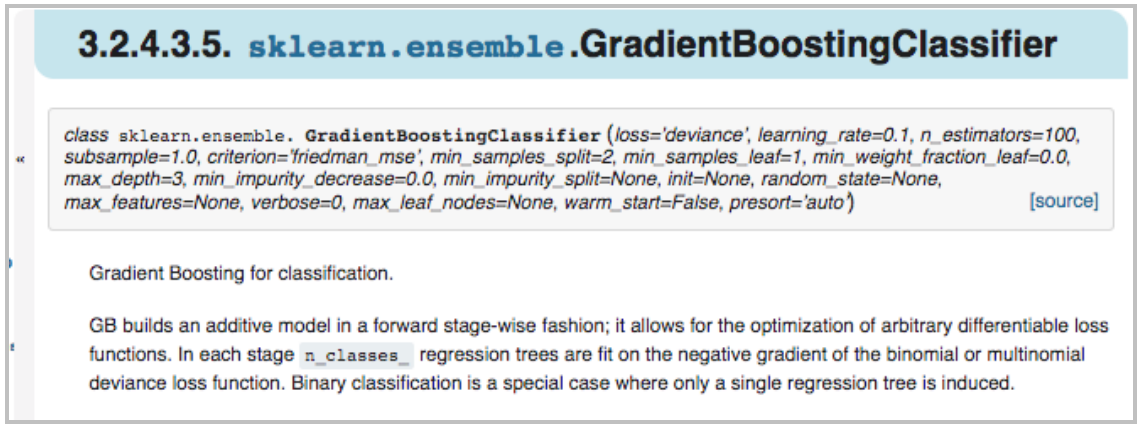
Knowledge Check 3

Module completion checklist

Objective	Complete
Introduce random forest and discuss use cases	✔
Summarize the concepts associated with random forest and bagging	✔
Transform Costa Rican data and implement random forest	✔
Introduce gradient boosting and how it compares to bagging	✔
Discuss gradient tree boosting within scikit-learn and implement on Costa Rican data	

scikit-learn - Gradient Tree Boosting

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



3.2.4.3.5. sklearn.ensemble.GradientBoostingClassifier

class sklearn.ensemble. **GradientBoostingClassifier** (*loss='deviance', learning_rate=0.1, n_estimators=100, subsample=1.0, criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_decrease=0.0, min_impurity_split=None, init=None, random_state=None, max_features=None, verbose=0, max_leaf_nodes=None, warm_start=False, presort='auto'*) [source]

Gradient Boosting for classification.

GB builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage `n_classes_` regression trees are fit on the negative gradient of the binomial or multinomial deviance loss function. Binary classification is a special case where only a single regression tree is induced.

- Gradient boosting builds an additive model in a sequential fashion
- It allows for the optimization of arbitrary differentiable loss functions
- In each stage, `n_classes_` regression trees are fit on the negative gradient of the binomial or multinomial deviance loss function
- Binary classification is a special case where only a single regression tree is induced.
- For all the parameters of the `GradientBoostingClassifier` package, visit ***scikit-learn's documentation***

GradientBoostingClassifier

- We just introduced the package `GradientBoostingClassifier`
- We are now going to implement gradient boosting on our cleaned data
- First, let's look at the **methods** available once the model is built

Methods

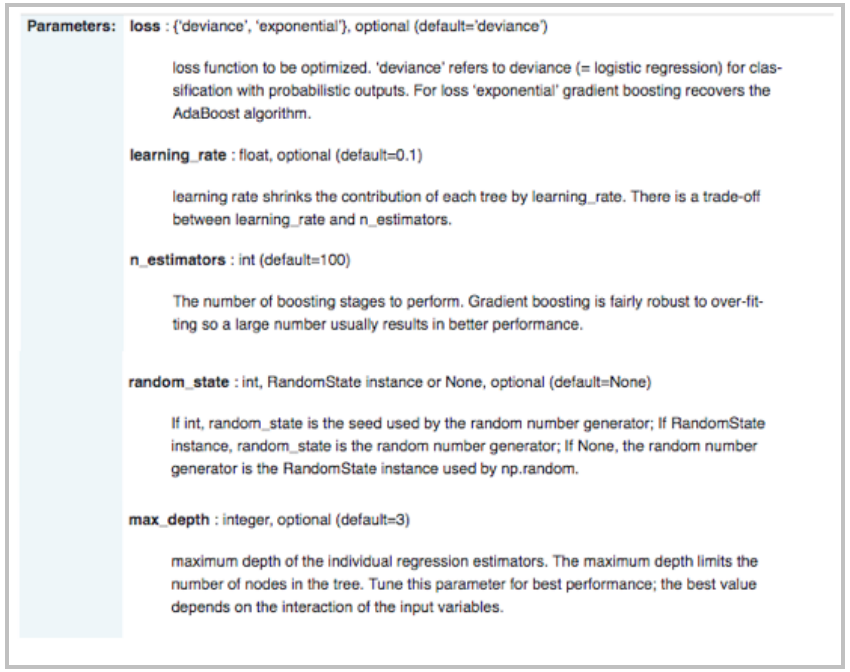
`apply` (X)	Apply trees in the ensemble to X, return leaf indices.
`decision_function` (X)	Compute the decision function of `x`.
`fit` (X, y[, sample_weight, monitor])	Fit the gradient boosting model.
`get_params` ([deep])	Get parameters for this estimator.
`predict` (X)	Predict class for X.
`predict_log_proba` (X)	Predict class log-probabilities for X.
`predict_proba` (X)	Predict class probabilities for X.
`score` (X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params` (**params)	Set the parameters of this estimator.
`staged_decision_function` (X)	Compute decision function of `x` for each iteration.
`staged_predict` (X)	Predict class at each stage for X.
`staged_predict_proba` (X)	Predict class probabilities at each stage for X.

- Your manager would like for you to build an additive model to optimize randomized differentiable loss functions by doing the following:
  - **Build** the gradient boosting model
  - **Fit** the model to the training data
  - **Predict** on the test data using our trained model
  - Store the predictions to revisit later today, using `pickle`

Boosting: build model

- **Build** the gradient boosting model

```
# Save the parameters we will be using for our gradient
boosting classifier.
gbm = GradientBoostingClassifier(n_estimators = 200,
learning_rate = 1,
max_depth = 2,
random_state = 1)
```



Parameters: **loss** : {'deviance', 'exponential'}, optional (default='deviance')

loss function to be optimized. 'deviance' refers to deviance (= logistic regression) for classification with probabilistic outputs. For loss 'exponential' gradient boosting recovers the AdaBoost algorithm.

learning_rate : float, optional (default=0.1)

learning rate shrinks the contribution of each tree by learning_rate. There is a trade-off between learning_rate and n_estimators.

n_estimators : int (default=100)

The number of boosting stages to perform. Gradient boosting is fairly robust to over-fitting so a large number usually results in better performance.

random_state : int, RandomState instance or None, optional (default=None)

If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random.

max_depth : integer, optional (default=3)

maximum depth of the individual regression estimators. The maximum depth limits the number of nodes in the tree. Tune this parameter for best performance; the best value depends on the interaction of the input variables.

Boosting: fit model

- **Fit** the model to the training data

```
# Fit the saved model to your training data.
gbm.fit(X_train, y_train)
```

```
GradientBoostingClassifier(criterion='friedman_mse', init=None, learning_rate=1,
                           loss='deviance', max_depth=2, 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,
                           n_estimators=200, n_iter_no_change=None,
                           presort='auto', random_state=1, subsample=1.0,
                           tol=0.0001, validation_fraction=0.1, verbose=0,
                           warm_start=False)
```

Boosting: predict

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

```
# Predict on test data.
predicted_values_gbm = gbm.predict(X_test)
print(predicted_values_gbm)
```

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

Confusion matrix and accuracy

Both `confusion_matrix` and `accuracy_score` take 2 arguments:

1. Original data labels
2. Predicted labels

```
# Take a look at test data confusion matrix.
conf_matrix_boosting = metrics.confusion_matrix(y_test, predicted_values_gbm)
print(conf_matrix_boosting)
```

```
[[1646  160]
 [ 229  833]]
```

```
# Compute test model accuracy score.
accuracy_gbm = metrics.accuracy_score(y_test, predicted_values_gbm)
print('Accuracy of gbm on test data: ', accuracy_gbm)
```

```
Accuracy of gbm on test data:  0.8643654114365411
```

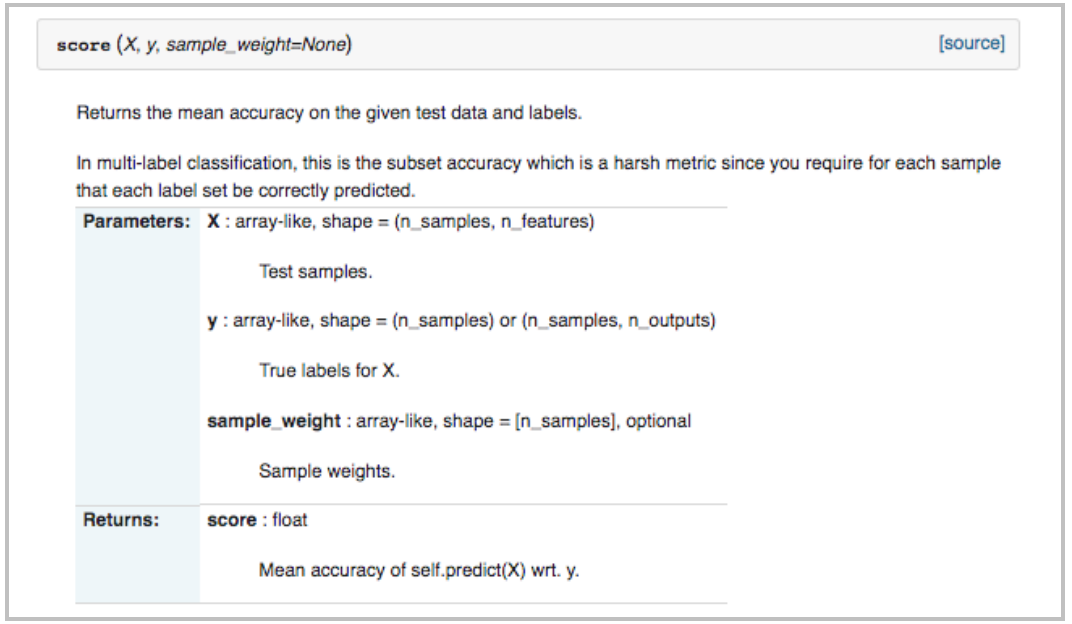
Accuracy of training model

- Let's look at the accuracy of the model we just built, on the training data

```
# Compute accuracy using training data.
train_accuracy_gbm = gbm.score(X_train, y_train)

print ("Train Accuracy:", train_accuracy_gbm)
```

```
Train Accuracy: 0.902975033637315
```

- 0.90 is high for accuracy
- Remember, this is accuracy on the **training** dataset
- It will be high, but this won't be the same result that you'll see on the **test** dataset



score (*X, y, sample_weight=None*) [source]

Returns the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters:

X : array-like, shape = (n_samples, n_features)

Test samples.

y : array-like, shape = (n_samples) or (n_samples, n_outputs)

True labels for X.

sample_weight : array-like, shape = [n_samples], optional

Sample weights.

Returns:

score : float

Mean accuracy of self.predict(X) wrt. y.

Pickle final accuracy

- Let's save our gradient boosting score in our `model_final` dataset
- **Finally, let's pickle our `model_final` dataset for next week**

```
# Add the model to our dataframe.
model_final_forest_gbm =
model_final_forest_gbm.append({'metrics' : "accuracy" ,

'values' : round(accuracy_gbm,4),

'model': 'boosting' } ,

ignore_index = True)
print(model_final_forest_gbm)
```

	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
9	accuracy	0.9338	random forest
10	accuracy	0.8644	boosting

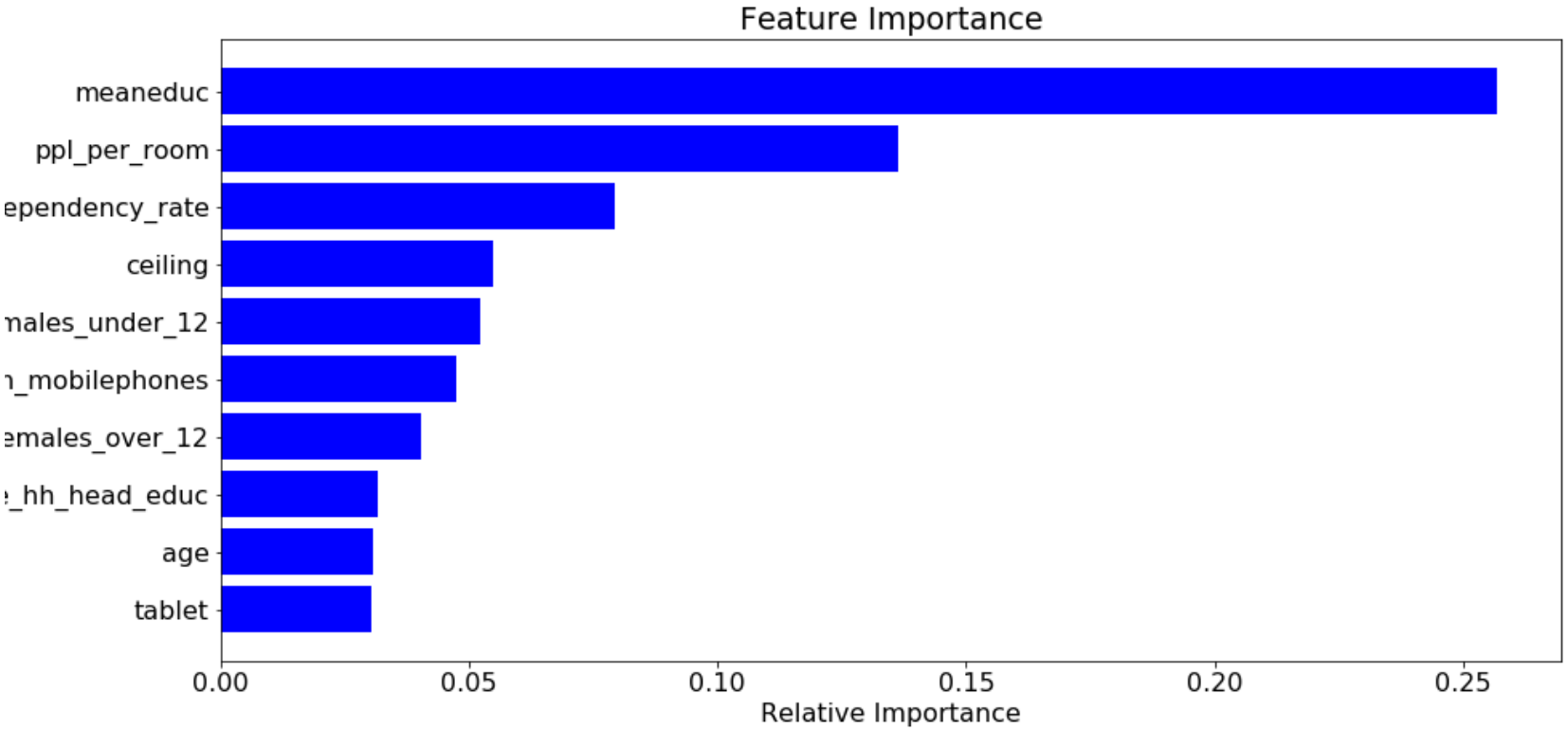
```
pickle.dump(model_final_forest_gbm,
open("model_final_forest_gbm.sav", "wb" ))
```

Our top 10 features

- Here is our feature importance plot for the gradient boosting model we just built
- We are looking at the top 10 features

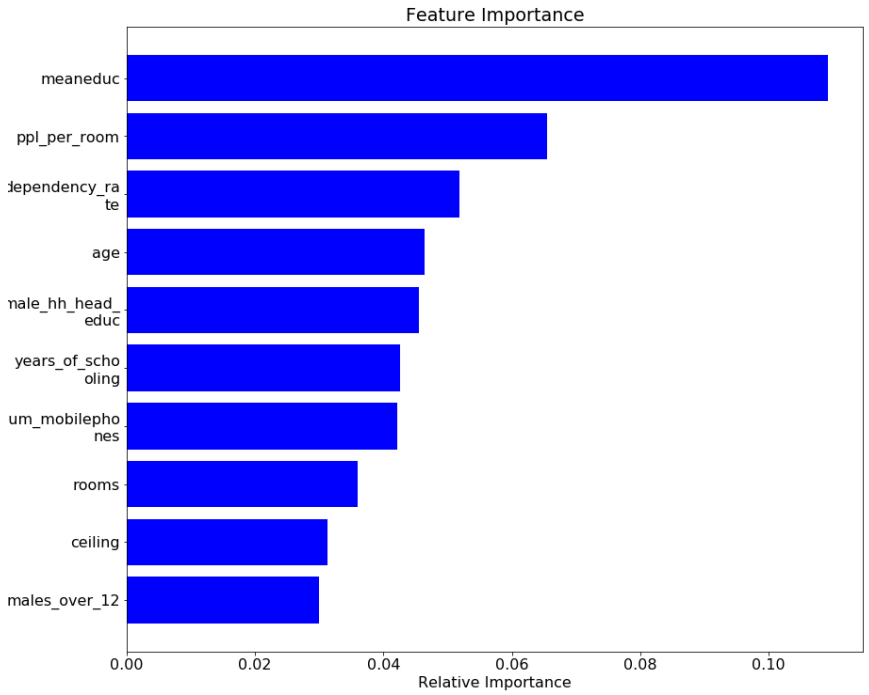
```
features = costarica_features.columns
importances = gbm.feature_importances_
indices = np. argsort(importances)[::-1]
top_indices = indices[0:10][::-1]

plt.figure(1)
plt.title('Feature Importance')
plt.barh(range(len(top_indices)), importances[top_indices], color = 'b', align = 'center')
labels = features[top_indices]
labels = [ '\n'.join(wrap(l,13)) for l in labels ]
plt.yticks(range(len(top_indices)), features[top_indices])
plt.xlabel('Relative Importance')
```

Feature importance plot

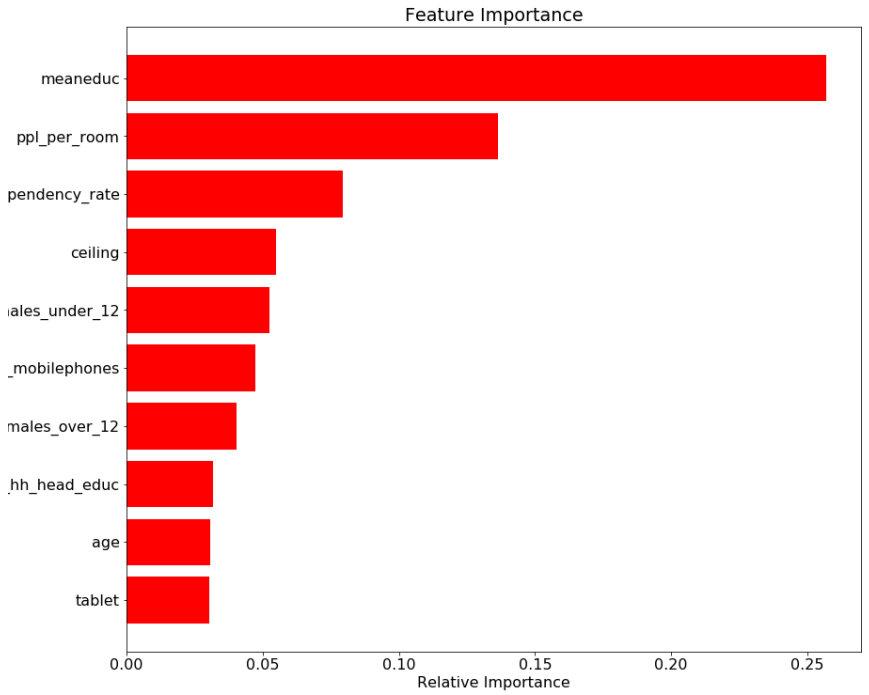


Compare feature importance plots

- **Random forest feature importance**



- **Gradient boosting feature importance**



Next steps: optimization of models

- Just like we did last class with decision trees, next class we will be **optimizing our ensemble methods**
- We will be using `GridSearchCV` and `RandomizedSearchCV` to optimize the models

Knowledge Check 4

Exercise 2

Module completion checklist

Objective	Complete
Introduce random forest and discuss use cases	✔
Summarize the concepts associated with random forest and bagging	✔
Transform Costa Rican data and implement random forest	✔
Introduce gradient boosting and how it compares to bagging	✔
Discuss gradient tree boosting within scikit-learn and implement on Costa Rican data	✔

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

- Implement ensemble methods on your dataset
- 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!