Advanced Classification - Part 2

One should look for what is and not what he thinks should be. (Albert Einstein)

Module completion checklist

Objective	Complete
Predict using random forest model and evaluate results	
Introduce gradient boosting and how it compares to bagging	
Discuss gradient tree boosting within scikit-learn and implement on Costa Rican data	

Loading packages

- Let's load the packages we will be using
- These packages are used for classification using gradient boosting and other tools

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

# Random forest and boosting packages
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
```

Recap: random forest

- What is a 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

Recap: 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 $p = number\ of\ predictors$

Datasets for today

- **We will be using the same datasets as earlier tiday, one for in-class practice and the other for self guided-exercise work**

- **A dataset in class to learn the concepts**
  - Costa Rica household poverty data by the Inter-American Development Bank

- **A dataset for our in-class exercises**
  - Bank marketing dataset
    - Related to direct marketing campaigns of a Portuguese banking institution
    - The target variable is whether the user subscribed to the bank term deposit (‘yes’) or not (‘no’)

Directory settings

- In order to maximize the efficiency of your workflow, you should encode your directory structure into `variables`
- We will use the `pathlib` library
- Let the `main_dir` be the variable corresponding to your `course` folder
- Let `data_dir` be the variable corresponding to your `data` folder

```
# Set 'main_dir' to location of the project folder
home_dir = Path(".").resolve()
main_dir = home_dir.parent.parent
print(main_dir)
```

```
data_dir = str(main_dir) + "/data"
print(data_dir)
```

Load the cleaned dataset

- Let's load the dataset from earlier today: `costa_clean`
- Assign it to `costa_clean` variable

```
costa_clean = pickle.load(open(data_dir + "/costa_clean.sav","rb"))
print(costa_clean.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]
```

Print info for our data

- Let's view the column names

```
costa_clean.columns
```

```
Index(['rooms', 'tablet', 'males_under_12', 'males_over_12', 'males_tot',
       'females_under_12', 'females_over_12', 'females_tot', 'ppl_under_12',
       'ppl_over_12', 'ppl_total', 'years_of_schooling', 'wall_block_brick',
       'wall_socket', 'wall_prefab_cement', 'wall_wood', 'floor_mos_cer_terr',
       'floor_cement', 'floor_wood', 'ceiling', 'electric_public',
       'electric_coop', 'toilet_sewer', 'toilet_septic', 'cookenergy_elec',
       'cookenergy_gas', 'trash_truck', 'trash_burn', 'wall_bad', 'wall_reg',
       'wall_good', 'roof_bad', 'roof_reg', 'roof_good', 'floor_bad',
       'floor_reg', 'floor_good', 'disabled_ppl', 'male', 'female', 'under10',
       'free', 'married', 'separated', 'single', 'hh_head', 'hh_spouse',
       'hh_child', 'num_child', 'num_adults', 'num_65plus', 'num_hh_total',
       'dependency_rate', 'male_hh_head_educ', 'female_hh_head_educ',
       'meaneduc', 'educ_none', 'educ_primary_inc', 'educ_primary',
       'educ_secondary_inc', 'educ_secondary', 'educ_undergrad', 'bedrooms',
       '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', 'rural_zone', 'age', 'Target'],
```

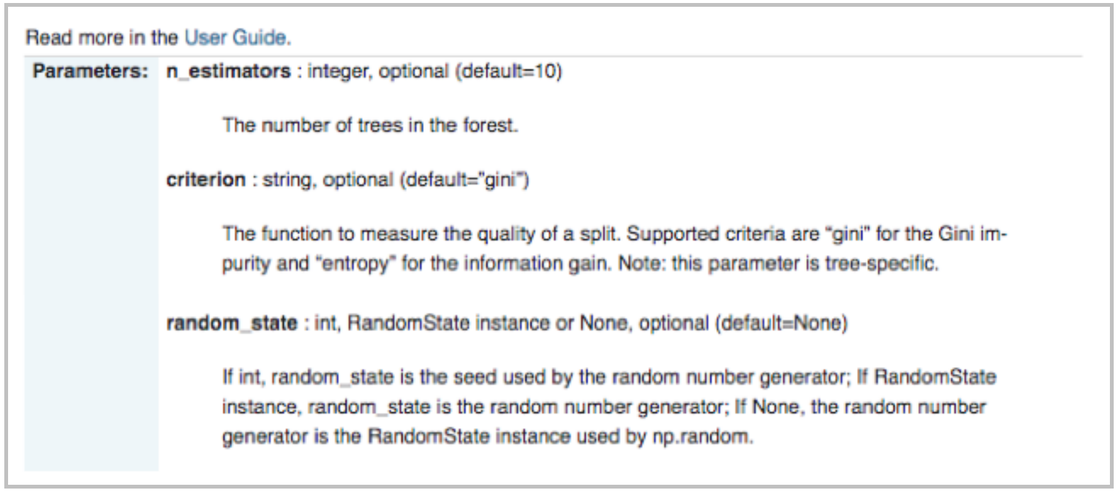
Split into training and test sets

```
# 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)
```

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(random_state=1)
```

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,])
```

```
[ True  True  True  True  True]
```

Confusion matrix: what is it

- A **confusion matrix** is what we use to measure error
- We use it to calculate Accuracy, Misclassification rate, True positive rate, False positive rate, and Specificity
- In the matrix overview of our data, let `Y1` be “non-vulnerable” and `Y2` be “vulnerable”

	Predicted Y1	**Predicted Y2**	**Actual totals**
Y1	**True Negative (TN)**	**False Positive (FP)**	Total negatives
Y2	**False Negative (FN)**	**True Positive (TP)**	Total positives
Predicted totals	Total predicted negatives	Total predicted positives	Total

Confusion matrix: summary

- Here is a table with all the metrics in one place:

Metric name	Formula
Accuracy	True positive + True Negative / Overall total
Misclassification rate	False positive + False Negative / Overall total
True positive rate	True positive / Actual yes (True positive + False negative)
False positive rate	False positive / Actual no (False positive + True negative)
Specificity	True negative / Actual no (False positive + True negative)

- We will use `Accuracy` to compare model performance in this module

Confusion matrix and accuracy

Both `confusion_matrix` and `accuracy_score` take **two** 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)
```

```
[[ 941  121]
 [  40 1766]]
```

```
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:  0.943863319386332
```

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 could mean that our model is overfitting.
- We saw that our accuracy on the **test** dataset is lower, at ~ `0.94`

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.

Evaluation of Random forest

- Let’s store the accuracy of this model:

```
# Create a dictionary with accuracy values for our\ random forest model.
model_final_dict = {'metrics': ["accuracy"],
                'values':[round(accuracy_forest,4)],
                 'model':['random_forest']}
model_final = pd.DataFrame(data = model_final_dict)
print(model_final)
```

```
    metrics  values          model
0  accuracy  0.9439  random_forest
```

- As we move forward, we will record our metrics for each new model we build
- This way, we can **compare** all classification models and **evaluate** which one seems to be the *model champion*

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, **illustrating** findings through **visualizations** can make them more accessible to your audience
- A **feature importance plot** will show the importance of the feature based on:
  - a decrease or an increase in rate of error; or
  - gain in impurity measure that is present when the feature is present

Applying feature importance on our data

- Suppose we would like 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 another person and potentially a larger audience

`feature_importances_` : *ndarray of shape (n_features,)*

Return the feature importances (the higher, the more important the feature).

Subsetting our features

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

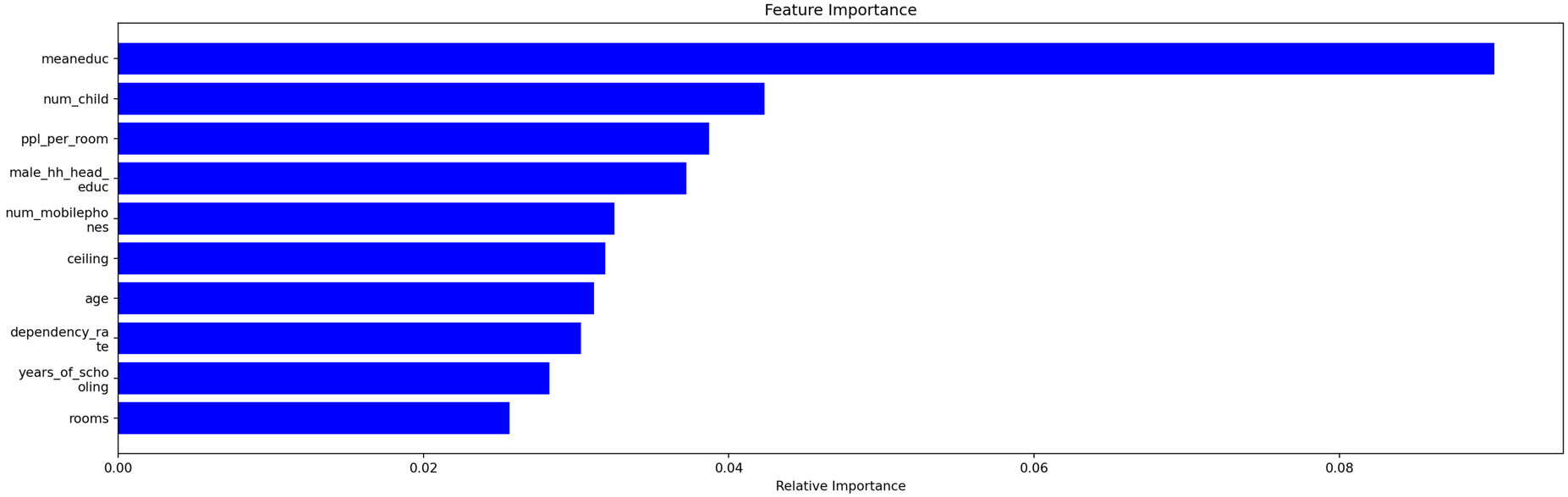
```
costarica_features = costa_clean.drop('Target', axis = 1)
```

- 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



Knowledge Check 1

Exercise 1

Module completion checklist

Objective	Complete
Predict using random forest model and evaluate results	✔
Introduce gradient boosting and how it compares to bagging	
Discuss gradient tree boosting within scikit-learn and implement on Costa Rican data	

Summary

- Today we learned about:
  - concept of Random forests
  - building a simple random forest model on our data
- In the next session, we will:
  - learn about gradient boosting model
  - optimize the random forest model and compare the optimized model to the base model

Warm up

- Before we start, check out this article on using boosted random forests to predict COVID-19 patient health: ***link***

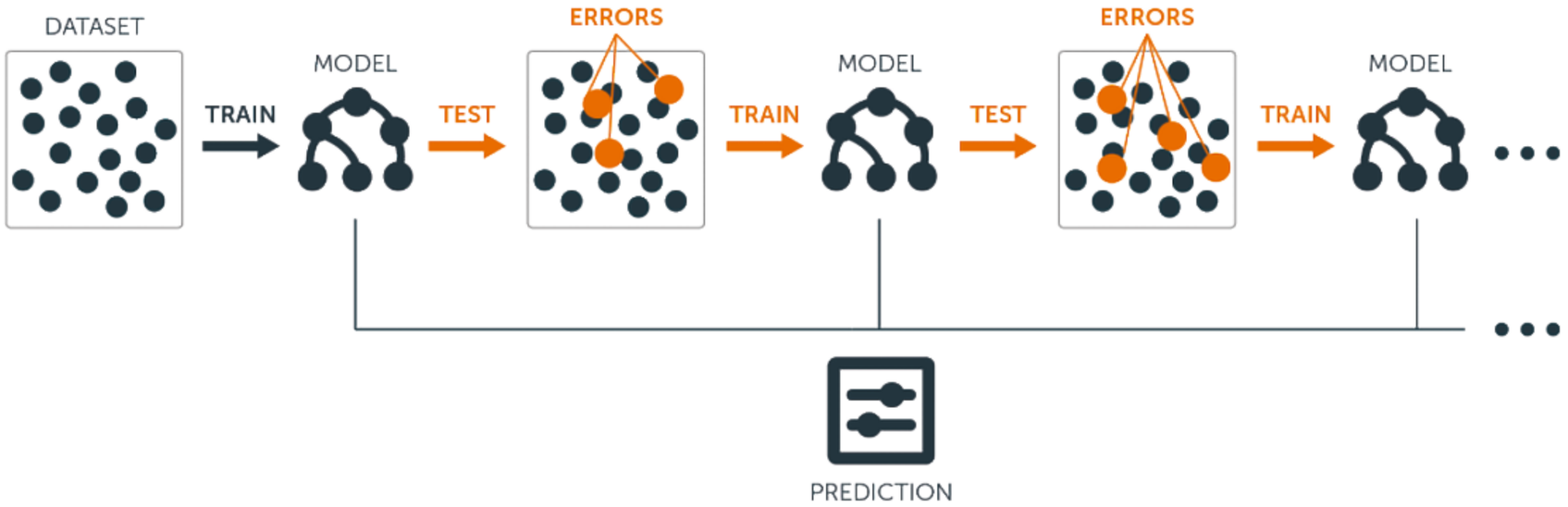
Welcome back

- In the previous module, we learned about random forest
- In this class, we will continue talking about random forests. Goals for today:
  - Introduce gradient boosting and how it compares to bagging
  - Implement various classification model performance metrics and discuss their interpretation
  - Optimize a random forest model to predict vulnerable households
  - Learn about optimizing gradient boosting models and compare their performance

Gradient boosted trees

- **Boosting**
  - Ensemble method, or 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



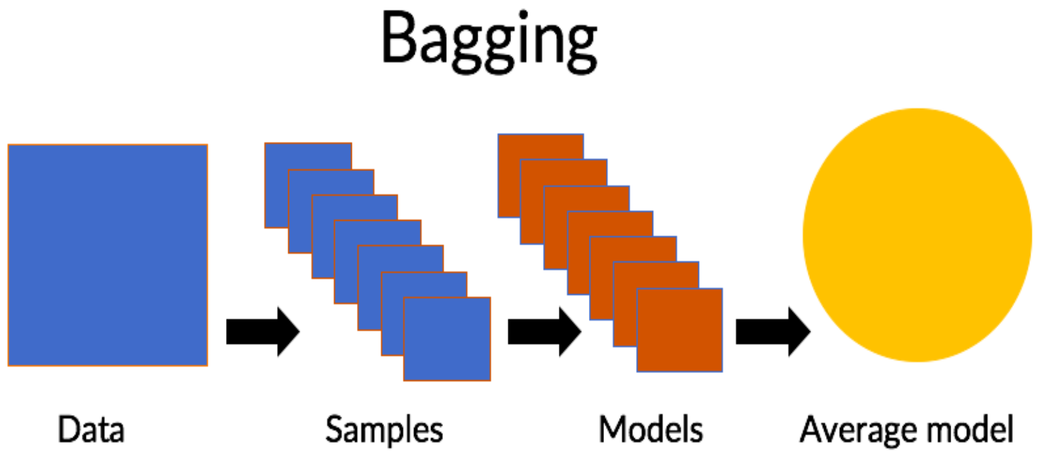
Source

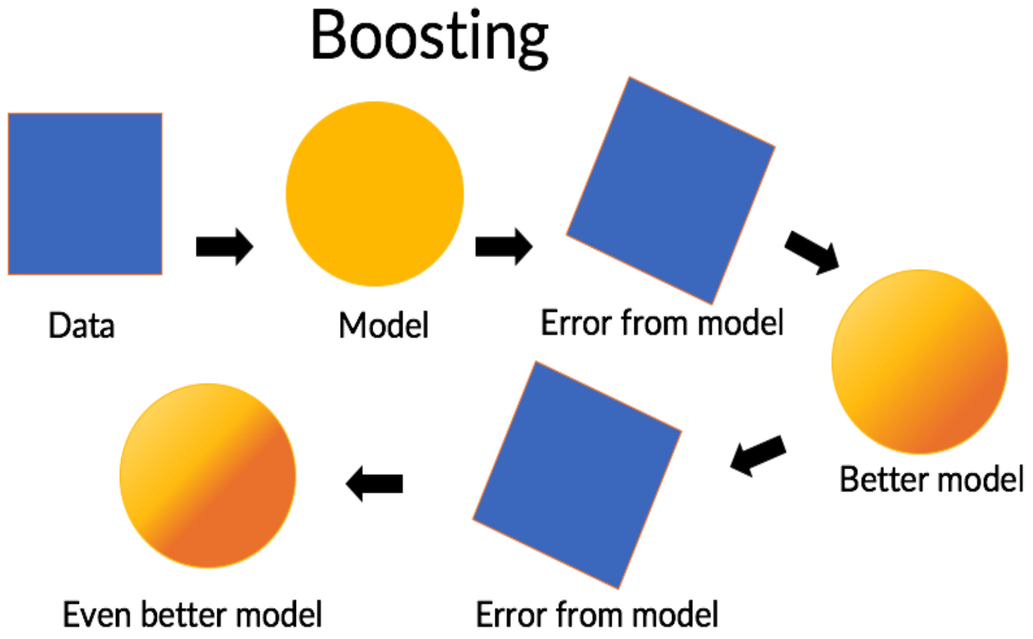
Boosting vs bagging

	Bagging	Boosting
Partitioning of the data into subsets	Random	Gives mis-classified 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:
  i. Fit a decision tree model to the data
  ii. Fit a decision tree model to the residuals
  iii. Create a new model

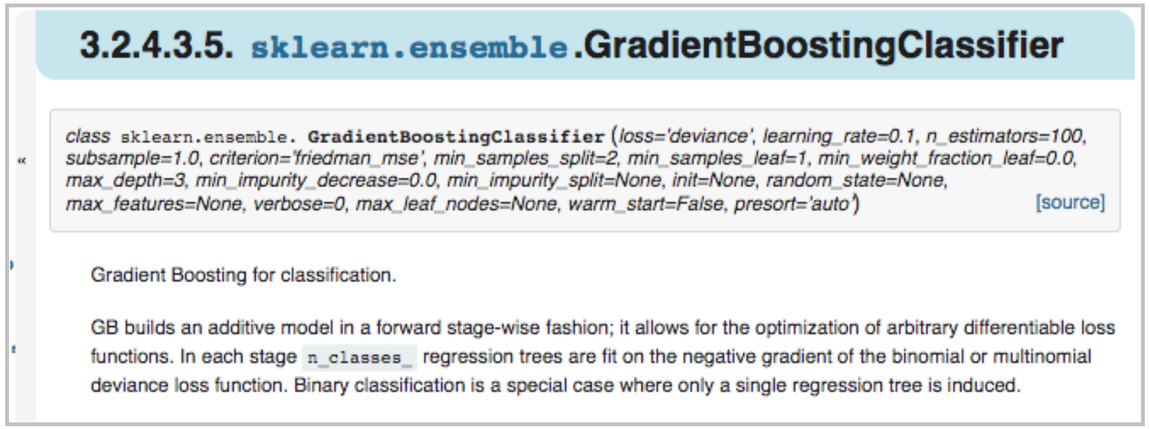
- **Gradient boosting** can be used with classification or regression
- The generalization of the multiple weak learners occurs by the optimization of a differentiable **loss function**
- The **loss function** will change based on the model's target variable:
  - **Regression**: ***gradient descent***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

Module completion checklist

Objective	Complete
Predict using random forest model and evaluate results	✔
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.

- We will perform the following steps:
  - **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 on, 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(learning_rate=1, max_depth=2, n_estimators=200,
                           random_state=1)
```

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)
```

```
[ True  True  True ...  True  True 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_boosting = metrics.confusion_matrix(y_test, predicted_values_gbm)
print(conf_matrix_boosting)
```

```
[[ 854  208]
 [ 155 1651]]
```

```
# 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:  0.8734309623430963
```

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: 0.9165794588129765
```

- Remember, this is accuracy on the **training** dataset
- It will be high, but the result that we saw on the **test** dataset, `0.87` is much lower

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
- We can see that the gbm model did not perform as well as random forest
- **We will run the vanilla models and optimize them in the next class!**

```
# Add the model to our dataframe.
model_final_forest_gbm = model_final.append(
                          {'metrics' : "accuracy" ,
                          'values' : round(accuracy_gbm,4),
                          'model': 'boosting' } ,
                          ignore_index = True)

print(model_final_forest_gbm)
```

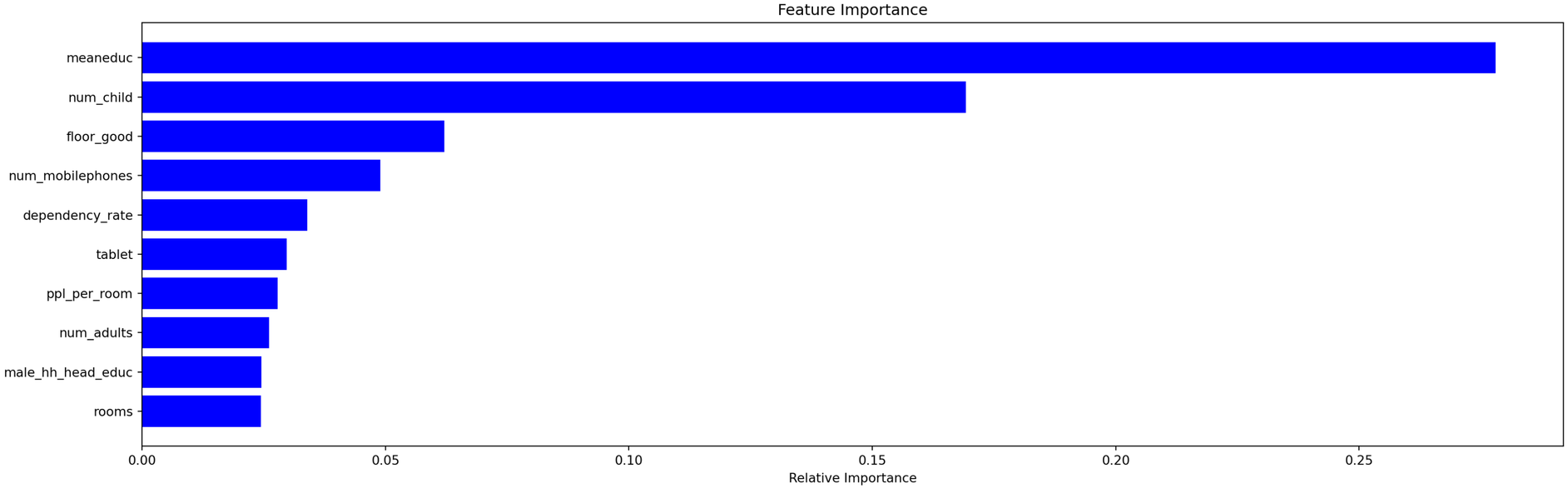
```
    metrics  values           model
0  accuracy  0.9439  random_forest
1  accuracy  0.8734        boosting
```

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

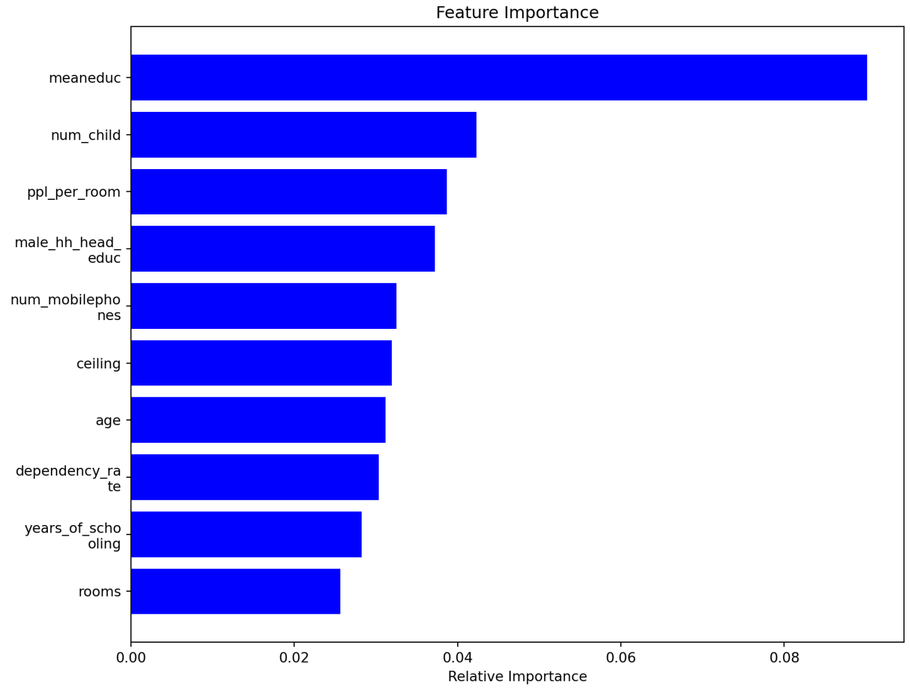
```
features = costarica_features.columns
importances = gbm.feature_importances_
indices = np.argsort(importances)[::-1]
top_indices = indices[0:10][::-1]

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

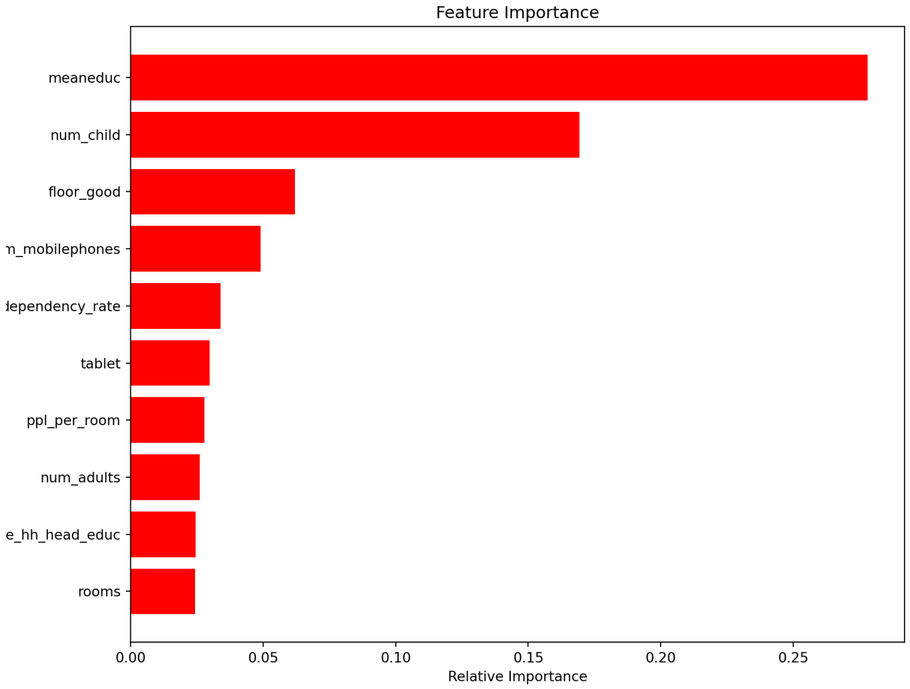


Compare feature importance plots

- **Random forest feature importance**



- **Gradient boosting feature importance**



- GBM plot looks different because of the entire training set being used vs OOB predictors

- In the next class we will be using various techniques to **optimize our ensemble methods**

Knowledge Check 2

Exercise 2

Module completion checklist

Objective	Complete
Predict using random forest model and evaluate results	✔
Introduce gradient boosting and how it compares to bagging	✔
Discuss gradient tree boosting within scikit-learn and implement on Costa Rican data	✔

Congratulations on completing this module!