DATA SOCIETY®

Advanced classification - ensemble methods

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

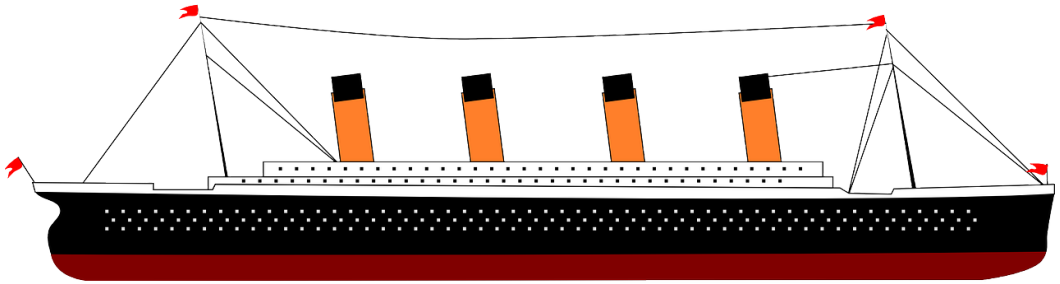
Welcome back!

- In the previous module, we learned:
  - When to use logistic regression
  - How to implement logistic regression on a training set and how to predict on a test set
  - How to analyze the results of logistic regression and tune the model using grid search cross-validation method

Homework recap

- As you know, **decision trees** predict the target value of an item by mapping observations about the item
- They 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

- For homework, you read ***here*** about how one data scientist applied decision trees to the Titanic dataset
  - *Could you follow along?*
  - *Did any of the results surprise you?*



Up Next: ensemble methods

- They improve machine learning results by combining several models
- This results in better predictive performance than a single model
- We'll cover two methods: **random forests** and **gradient boosting**

Module completion checklist

Objective	Complete
Introduce random forests and discuss use cases	
Summarize the concepts associated with random forests and bagging	
Prepare to implement a random forest	
Implement random forests on the dataset	
Introduce gradient boosting and how it compares to bagging	
Discuss gradient tree boosting within scikit-learn and implement on credit card data	

Random forests

- What is a random forest?
  - It is an ensemble method used for **classification and regression tasks**
  - It's a supervised learning algorithm which builds **multiple decision trees** and aggregates the result
  - It uses a technique called bootstrap aggregation, commonly known as **bagging**
  - It limits overfitting and bias error

Random forests: use cases

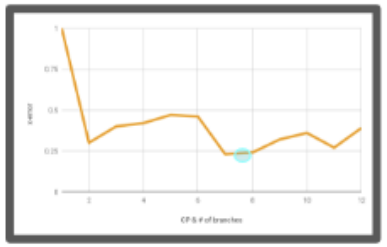
- The random forests algorithm is used in a multitude of industries such as banking, medicine, and e-commerce
- For example, the random forests algorithm can be used in:
  - fraud detection
  - identifying a disease by analyzing a patient's medical history
  - predicting the behavior of the stock market
  - understanding whether a customer will buy a product, or not

Decision tree process

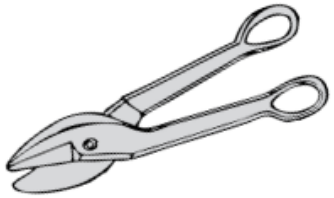
- You already know that decision trees **predict** the target value of an item by **mapping observations about the item**
- So why would we use a random forests?

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 -			

Random forests vs. decision trees

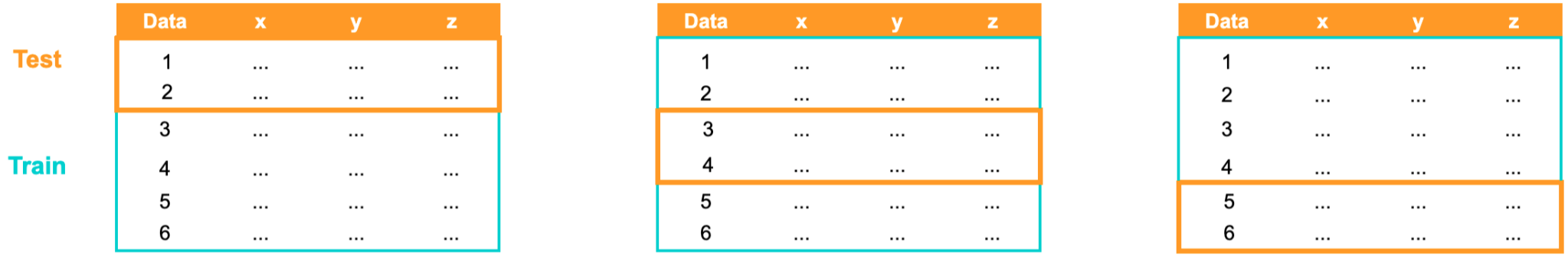
- Use **random forests** over decision trees to:
  - reduce overfitting
  - get higher predictive accuracy
  - be more efficient with large datasets
- Why would we use **decision trees** instead?
  - They are intuitive and give easily interpretable results
  - It's a less computationally expensive algorithm

Module completion checklist

Objective	Complete
Introduce random forests and discuss use cases	✔
Summarize the concepts associated with random forests and bagging	
Prepare to implement a random forest	
Implement random forests on the dataset	
Introduce gradient boosting and how it compares to bagging	
Discuss gradient tree boosting within scikit-learn and implement on credit card data	

Why is random forests 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 forests of trees is used as a metric to assess the algorithms' performance
- It uses a built-in form of the 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 decision trees lets you choose **how many trees**, i.e. how many bootstrapped sampled training sets to create

Random forests: bagging

Bagging of observations in itself is pretty cool!

But that's not all it does...

Bagging is not enough

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

Obs	X1	X2	Y1	Y2
1	2.5	3.6	4.8	3.7
2	2.8	4.7	-2.8	7.1
3	5.8	9.7	9.1	13

Obs	X1	X2	Y1	Y2
2	2.8	4.7	-2.8	7.1
1	2.5	3.6	4.8	3.7
3	5.8	9.7	9.1	13

Obs	X1	X2	Y1	Y2
3	5.8	9.7	9.1	13
1	2.5	3.6	4.8	3.7
2	2.8	4.7	-2.8	7.1

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 forests** are:

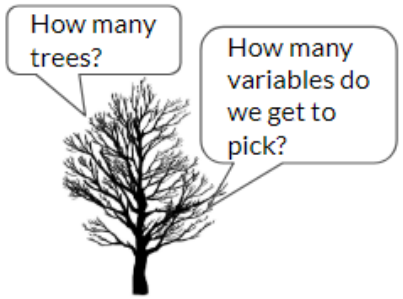
1. number of trees
2. number of features per tree

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 $p = number\ of\ predictors$

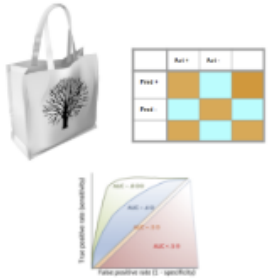
Random forests methodology

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

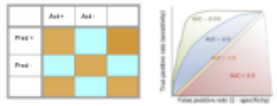


Step 2:
Build forest on training data

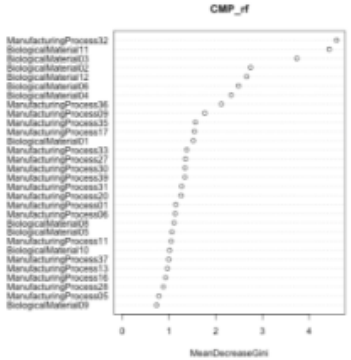
Step 3:
Check performance



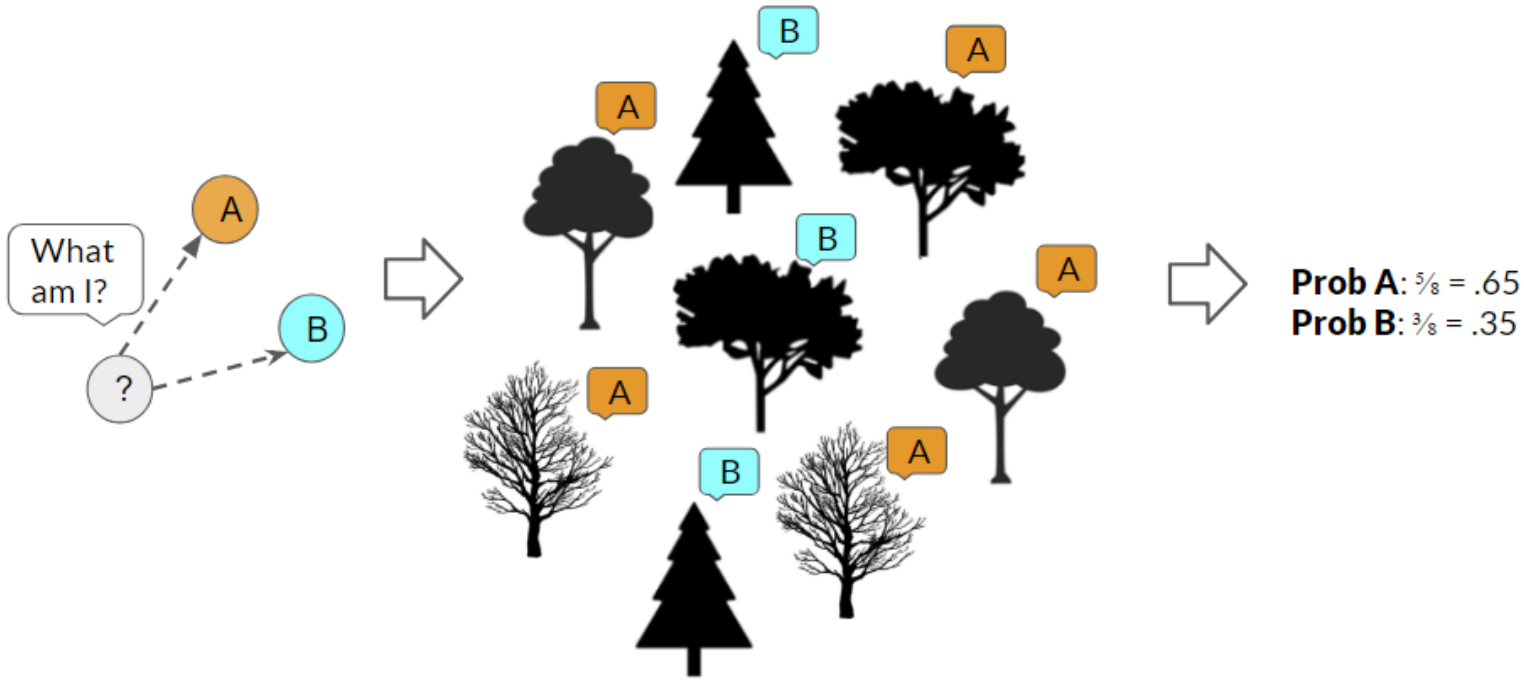
Step 4:
Apply to test data & evaluate



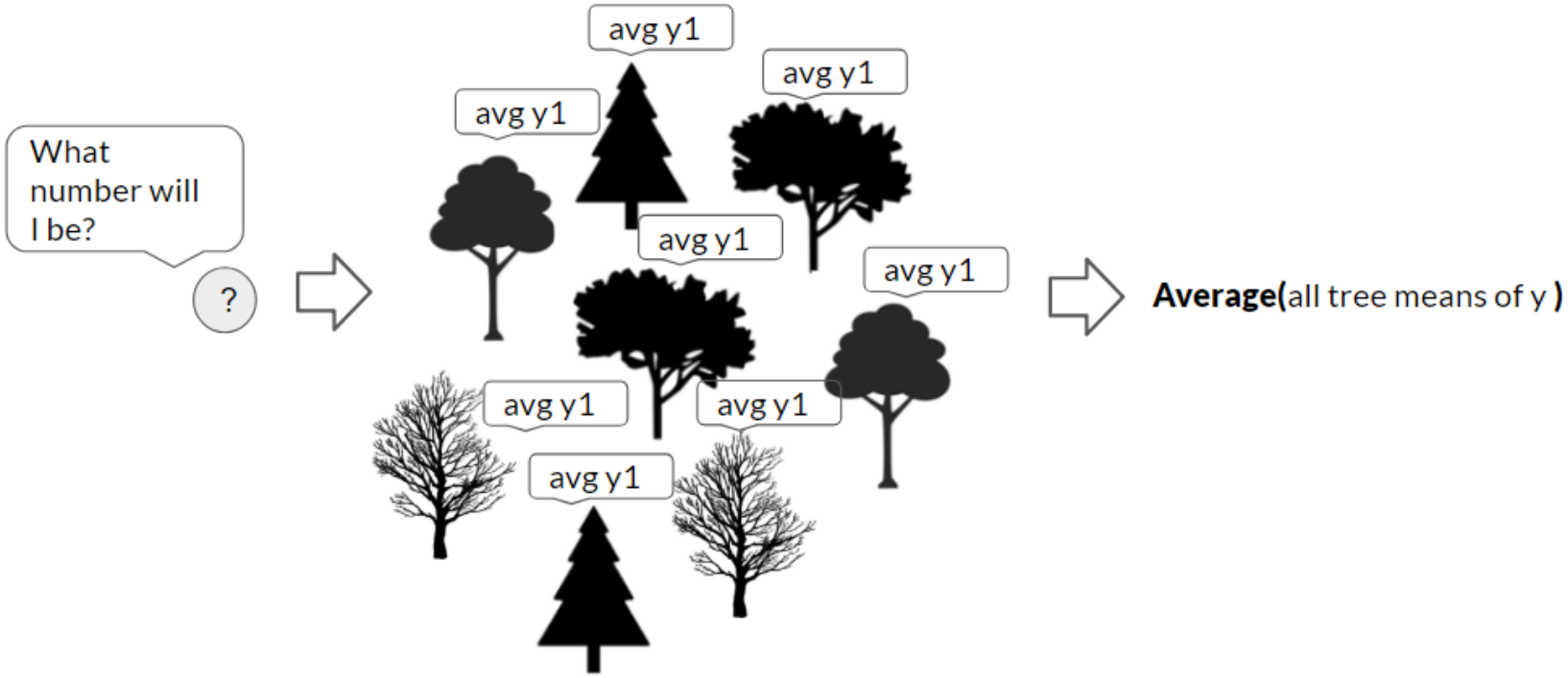
Step 5:
Use variable importance as needed



Random forests classification



Handling regression with random forests



For this module, we will focus on classification

Module completion checklist

Objective	Complete
Introduce random forests and discuss use cases	✔
Summarize the concepts associated with random forests and bagging	✔
Prepare to implement a random forest	
Implement random forests on the dataset	
Introduce gradient boosting and how it compares to bagging	
Discuss gradient tree boosting within scikit-learn and implement on credit card 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 `advanced-classification` folder

```
from pathlib import Path
# Set `home_dir` to the root directory of your computer.
home_dir = Path.home()

# Set `main_dir` to the location of your `advanced-classification` folder.
main_dir = home_dir / "Desktop" / "advanced-classification"

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

Working directory

- Set working directory to the `data_dir` variable we set

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

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

```
/home/[user-name]/Desktop/advanced-classification/data
```

Loading packages

- Load the packages we will be using

```
# Helper packages.
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from textwrap import wrap
import pickle
```

```
# Model set up and tuning packages from scikit-learn.
from sklearn.model_selection import train_test_split

# Scikit-learn package for data preprocessing.
from sklearn import preprocessing

# Scikit-learn packages for evaluating model performance.
from sklearn import metrics
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
```

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

Loading data into Python

- Let's load the entire `credit_card_data` dataset using the `read_csv` function

```
credit_card = pd.read_csv("credit_card_data.csv")
print(credit_card.head())
```

```
   ID  LIMIT_BAL  SEX  ...  PAY_AMT5  PAY_AMT6  default_payment_next_month
0   1      20000    2  ...         0         0                          1
1   2     120000    2  ...         0      2000                          1
2   3      90000    2  ...      1000      5000                          0
3   4      50000    2  ...      1069      1000                          0
4   5      50000    1  ...       689       679                          0

[5 rows x 25 columns]
```

Renaming target variable

- Rename the target variable to 'default_payment'

```
credit_card = credit_card.rename(columns = {'default_payment_next_month' : 'default_payment'})
print(credit_card.head())
```

```
   ID  LIMIT_BAL  SEX  EDUCATION  ...  PAY_AMT4  PAY_AMT5  PAY_AMT6  default_payment
0   1      20000    2          2  ...         0         0         0                1
1   2     120000    2          2  ...      1000         0      2000                1
2   3      90000    2          2  ...      1000      1000      5000                0
3   4      50000    2          2  ...      1100      1069      1000                0
4   5      50000    1          2  ...      9000       689       679                0

[5 rows x 25 columns]
```

The data at first glance

- Look at the first 5 rows and the data types

```
# The first 5 rows.
print(credit_card.head())
```

```
   ID  LIMIT_BAL  SEX  EDUCATION  ...  PAY_AMT4  PAY_AMT5
PAY_AMT6  default_payment
0   1      20000    2          2  ...         0         0
0                 1
1   2     120000    2          2  ...      1000         0
2000              1
2   3      90000    2          2  ...      1000      1000
5000              0
3   4      50000    2          2  ...      1100      1069
1000              0
4   5      50000    1          2  ...      9000       689
679               0

[5 rows x 25 columns]
```

```
# The data types.
print(credit_card.dtypes)
```

```
ID                    int64
LIMIT_BAL             int64
SEX                   int64
EDUCATION             int64
MARRIAGE              int64
AGE                   int64
PAY_0                 int64
PAY_2                 int64
PAY_3                 int64
PAY_4                 int64
PAY_5                 int64
PAY_6                 int64
BILL_AMT1           float64
BILL_AMT2             int64
BILL_AMT3             int64
BILL_AMT4             int64
BILL_AMT5             int64
BILL_AMT6             int64
PAY_AMT1              int64
PAY_AMT2              int64
PAY_AMT3              int64
PAY_AMT4              int64
PAY_AMT5              int64
PAY_AMT6              int64
default_payment       int64
dtype: object
```

Frequency table of the target variable

- Now let's check the frequency of 'default_payment'

```
print(credit_card['default_payment'].value_counts())
```

```
0    23364
1     6636
Name: default_payment, dtype: int64
```

- It has **two levels**, 0 and 1

Data prep: check for NAs

- Check for NAs

```
# Check for NAs.
print(credit_card.isnull().sum())
```

- We have 1 missing value in the variable column 'BILL_AMT1'

```
ID                  0
LIMIT_BAL           0
SEX                 0
EDUCATION           0
MARRIAGE            0
AGE                 0
PAY_0               0
PAY_2               0
PAY_3               0
PAY_4               0
PAY_5               0
PAY_6               0
BILL_AMT1           1
BILL_AMT2           0
BILL_AMT3           0
BILL_AMT4           0
BILL_AMT5           0
BILL_AMT6           0
PAY_AMT1            0
PAY_AMT2            0
PAY_AMT3            0
PAY_AMT4            0
PAY_AMT5            0
PAY_AMT6            0
default_payment     0
dtype: int64
```

Filling missing values

- We will fill the missing value in 'BILL_AMT1' with the mean value

```
 # Fill missing values with mean
credit_card = credit_card.fillna(credit_card.mean()['BILL_AMT1'])
```

```
 # Check for NAs in 'BILL_AMT1'.
print(credit_card.isnull().sum()['BILL_AMT1'])
```

```
0
```

- We do not have any NAs now; we are ready to scale our predictors!

Transform categorical variables into dummies

- Let's transform the categorical variables into dummy variables and save it into a dataframe

```
# Convert 'sex' into dummy variables.
sex_dummy = pd.get_dummies(credit_card['SEX'], prefix = 'sex', drop_first = True)
# Convert 'education' into dummy variables.
education_dummy = pd.get_dummies(credit_card['EDUCATION'], prefix = 'education', drop_first = True)
# Convert 'marriage' into dummy variables.
marriage_dummy = pd.get_dummies(credit_card['MARRIAGE'], prefix = 'marriage', drop_first = True)
```

```
# Drop `sex`, `education`, `marriage` from the data.
credit_card.drop(['SEX', 'EDUCATION', 'MARRIAGE'], axis = 1, inplace = True)
```

```
# Concatenate `sex_dummy`, `education_dummy`, `marriage_dummy` to our dataset.
credit_card = pd.concat([credit_card,sex_dummy,education_dummy,marriage_dummy],axis=1)
print(credit_card.head())
```

```
   ID  LIMIT_BAL  AGE  PAY_0  ...  education_6  marriage_1  marriage_2  marriage_3
0   1      20000   24      2  ...           0           1           0           0
1   2     120000   26     -1  ...           0           0           1           0
2   3      90000   34      0  ...           0           0           1           0
3   4      50000   37      0  ...           0           1           0           0
4   5      50000   57     -1  ...           0           1           0           0

[5 rows x 32 columns]
```

Data prep: ready for random forests

- 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(credit_card.default_payment.dtypes)
```

```
int64
```

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

```
credit_card["default_payment"] = np.where(credit_card["default_payment"] == 1, True, False)

# Check class again.
print(credit_card.default_payment.dtypes)
```

```
bool
```

- Lastly, let's drop the unnecessary identifiers from the dataset

```
#dropping unnecessary identifier 'ID'
credit_card = credit_card.drop('ID',axis = 1)
```

Knowledge check 1

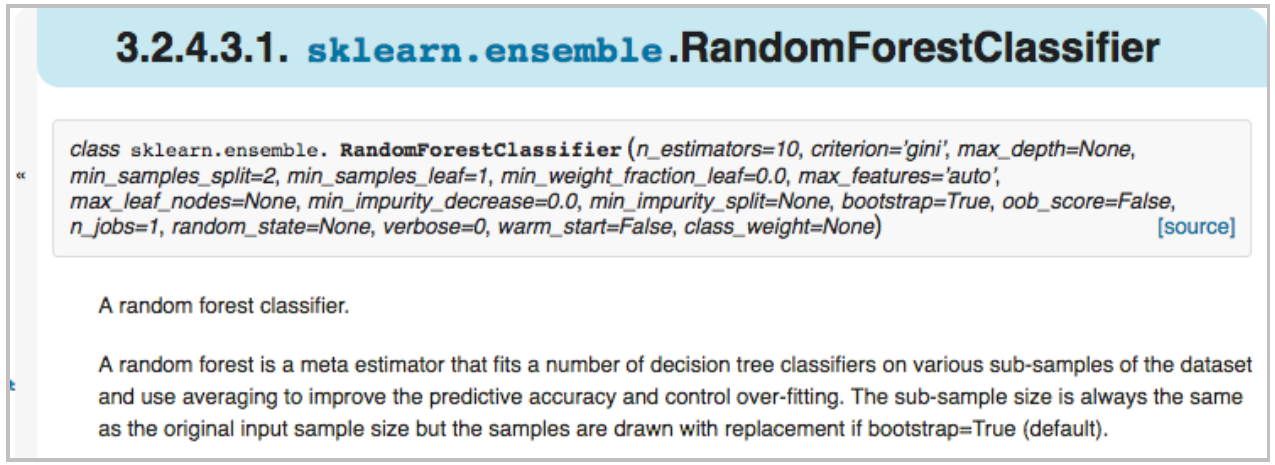
Exercise 1

Module completion checklist

Objective	Complete
Introduce random forests and discuss use cases	✔
Summarize the concepts associated with random forests and bagging	✔
Prepare to implement a random forest	✔
Implement random forests on the dataset	
Introduce gradient boosting and how it compares to bagging	
Discuss gradient tree boosting within scikit-learn and implement on credit card data	

Scikit-learn: random forests

- 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***

Split into training and test sets

- Now we split our data into training and test sets
- We run a decision tree initially on the training data

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

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

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

RandomForestClassifier

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

Methods

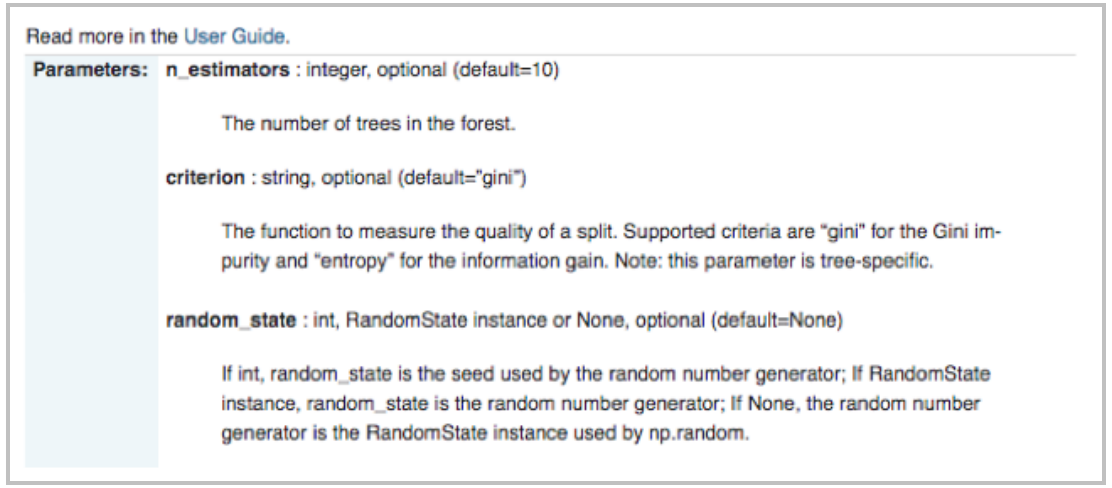
`apply` (X)	Apply trees in the forest to X, return leaf indices.
`decision_path` (X)	Return the decision path in the forest
`fit` (X, y[, sample_weight])	Build a forest of trees from the training set (X, y).
`get_params` ([deep])	Get parameters for this estimator.
`predict` (X)	Predict class for X.
`predict_log_proba` (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 forests 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 forests 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,])
```

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

Evaluate model

- By now, we are familiar with classification metrics for evaluating a model
- We can **use the same metrics we have discussed already** to measure how well our simple decision tree is doing:
  - Confusion matrix
  - ROC
  - AUC

Confusion matrix and accuracy

Remember, 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)
```

```
[[6591  409]
 [1288  712]]
```

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

```
Accuracy for random forests on test data:  0.8114444444444444
```

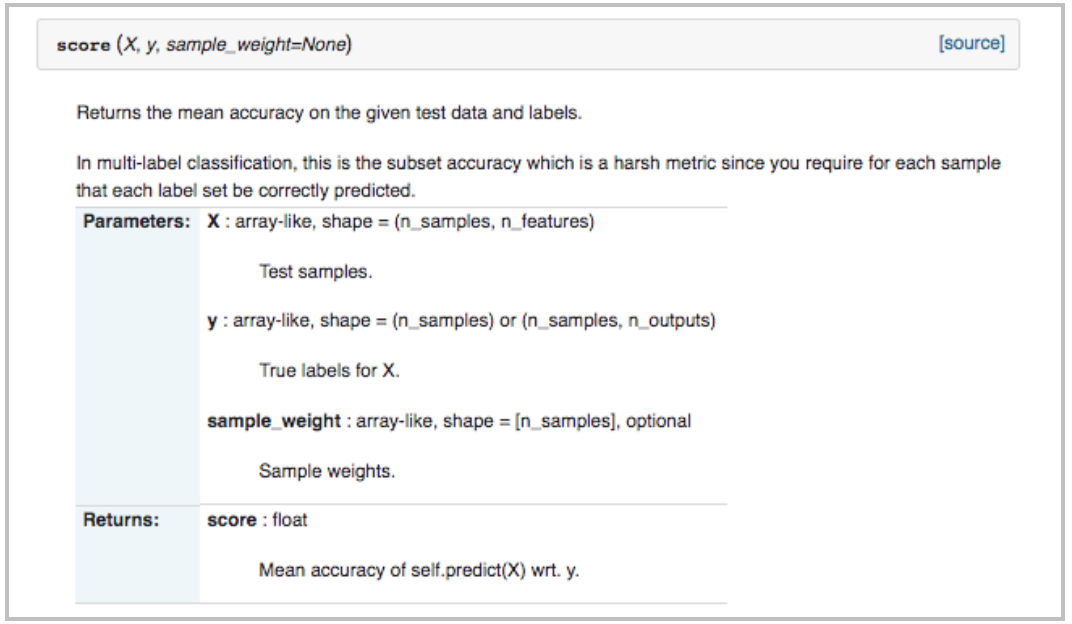
Accuracy of the training dataset

- 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: 0.9993333333333333
```

- 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.

Evaluation of random forests

- Let's load the pickled `model_final` dataframe and append the results to it

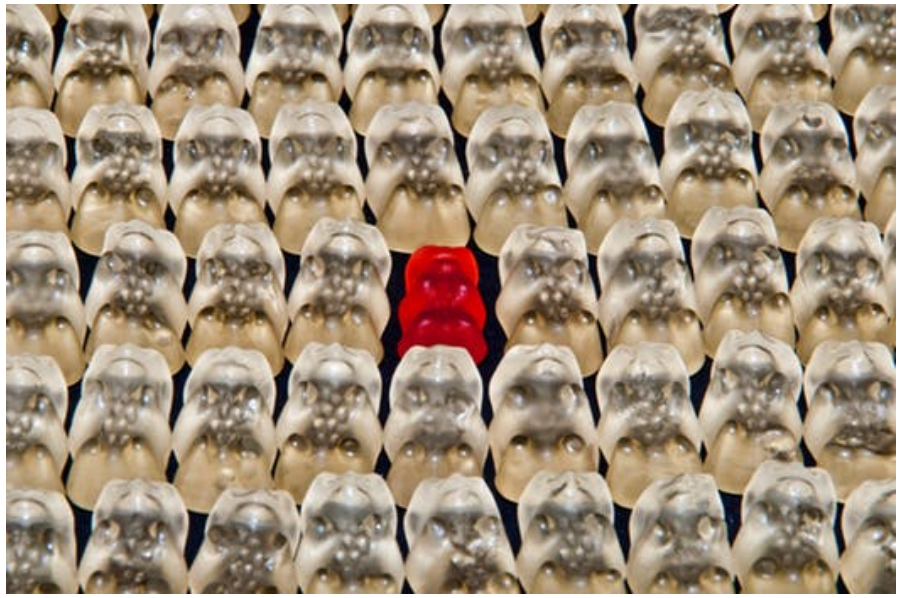
```
model_final = pickle.load(open("model_final.sav","rb"))
```

```
#Add this model to our model champion dataframe.
model_final = model_final.append({'metrics': "accuracy",
                'values':round(accuracy_forest,4),
                 'model':'random_forest'},
                 ignore_index = True)

print(model_final)
```

```
    metrics  values                   model
0  accuracy  0.7778                logistic
1  accuracy  0.7778  logistic_whole_dataset
2  accuracy  0.8077          logistic_tuned
3  accuracy  0.8114           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, **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

> **`feature_importances_`** : ***ndarray of shape (n_features,)***
>
> Return the feature importances (the higher, the more important the feature).

- 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
- Doing so will allow for another person, or a larger audience, to more clearly understand our algorithm

Subsetting our features

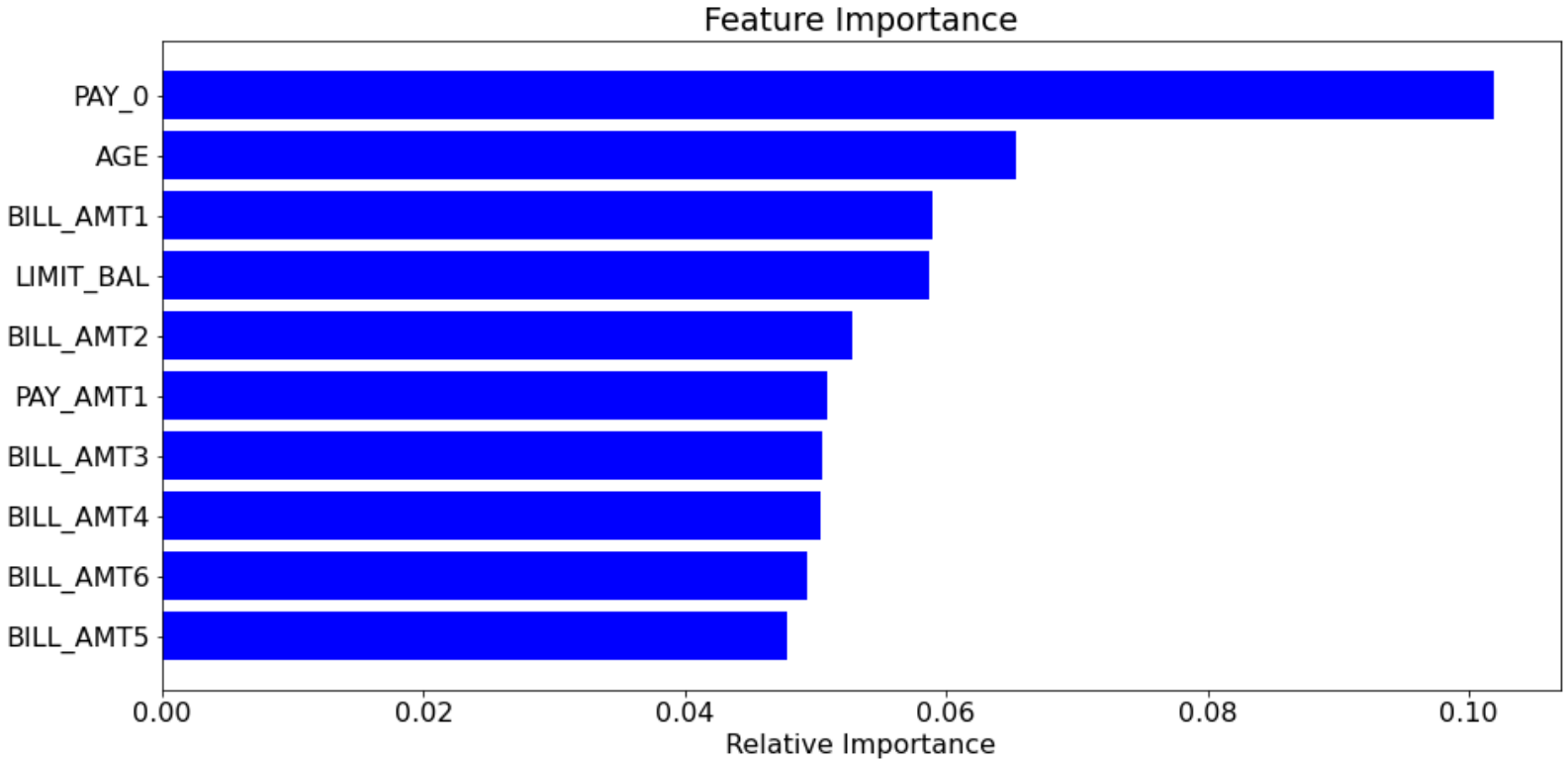
- Let's subset the features into another variable named `credit_card_features`

```
credit_card_features = credit_card.drop('default_payment', axis = 1)
```

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

```
features = credit_card_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')
```

Feature importance plot



- What can you tell about this plot?
- Why do you think `PAY_0` and `AGE` appear as variables with a lot of "importance"?

Knowledge check 2

Exercise 2

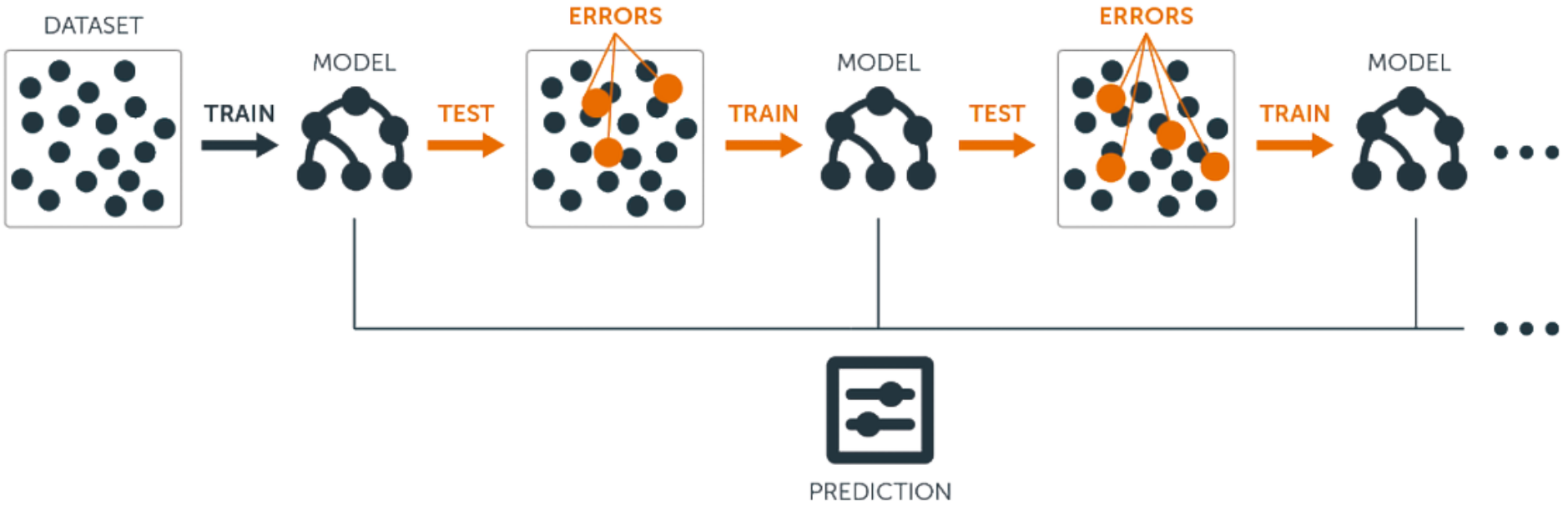
Module completion checklist

Objective	Complete
Introduce random forests and discuss use cases	✔
Summarize the concepts associated with random forests and bagging	✔
Prepare to implement a random forest	✔
Implement random forests on the dataset	✔
Introduce gradient boosting and how it compares to bagging	
Discuss gradient tree boosting within scikit-learn and implement on credit card data	

Gradient boosting

- **Gradient boosting** is an 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

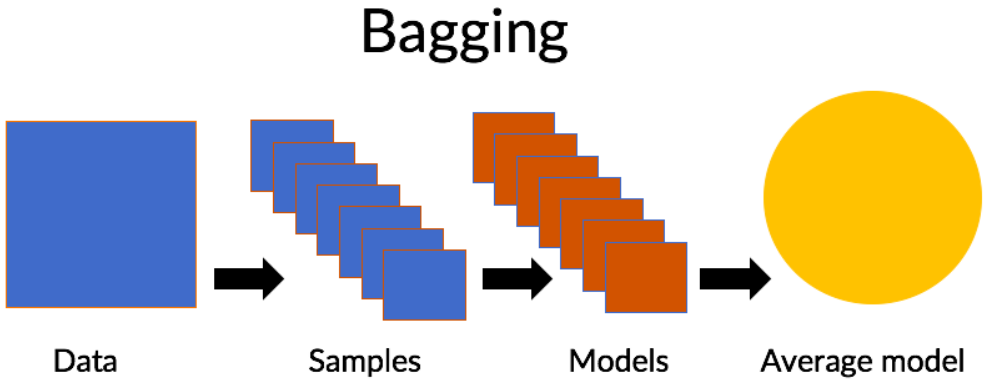


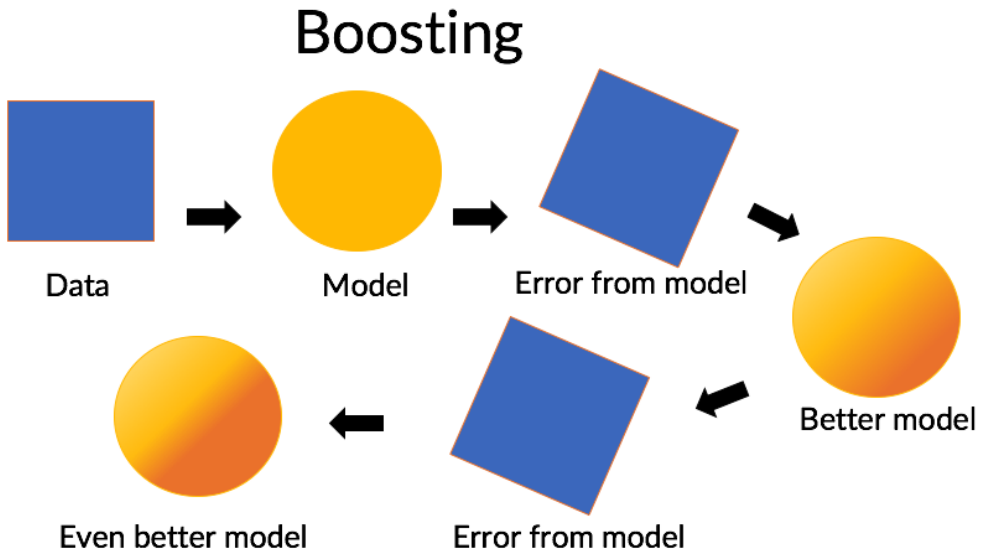
Source

Boosting vs bagging

	Bagging	Boosting
Partitioning of the data into subsets	Random	Gives 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***

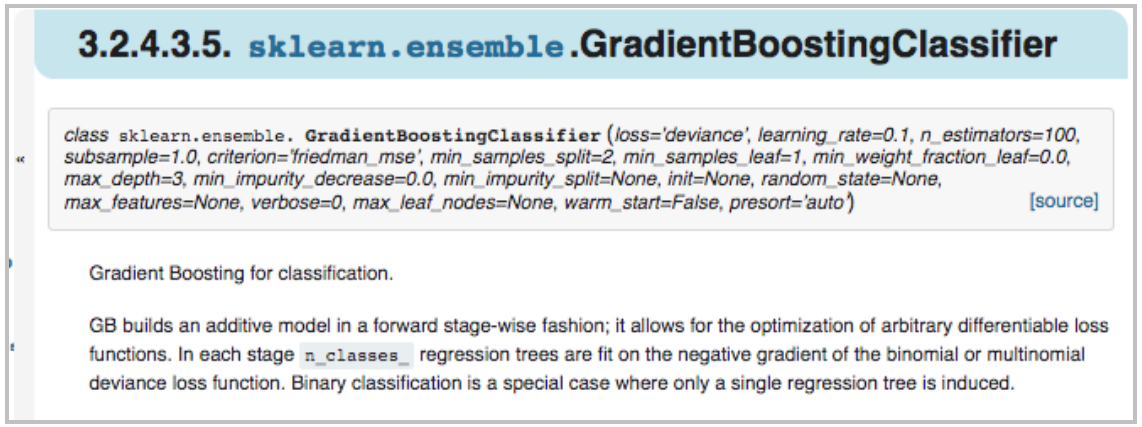
Knowledge check 3

Module completion checklist

Objective	Complete
Introduce random forests and discuss use cases	✔
Summarize the concepts associated with random forests and bagging	✔
Prepare to implement a random forest	✔
Implement random forests on the dataset	✔
Introduce gradient boosting and how it compares to bagging	✔
Discuss gradient tree boosting within scikit-learn and implement on credit card 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 are now going to use `GradientBoostingClassifier` 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)
```

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

```
[[6533  467]
 [1268  732]]
```

```
# 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:  0.8072222222222222
```

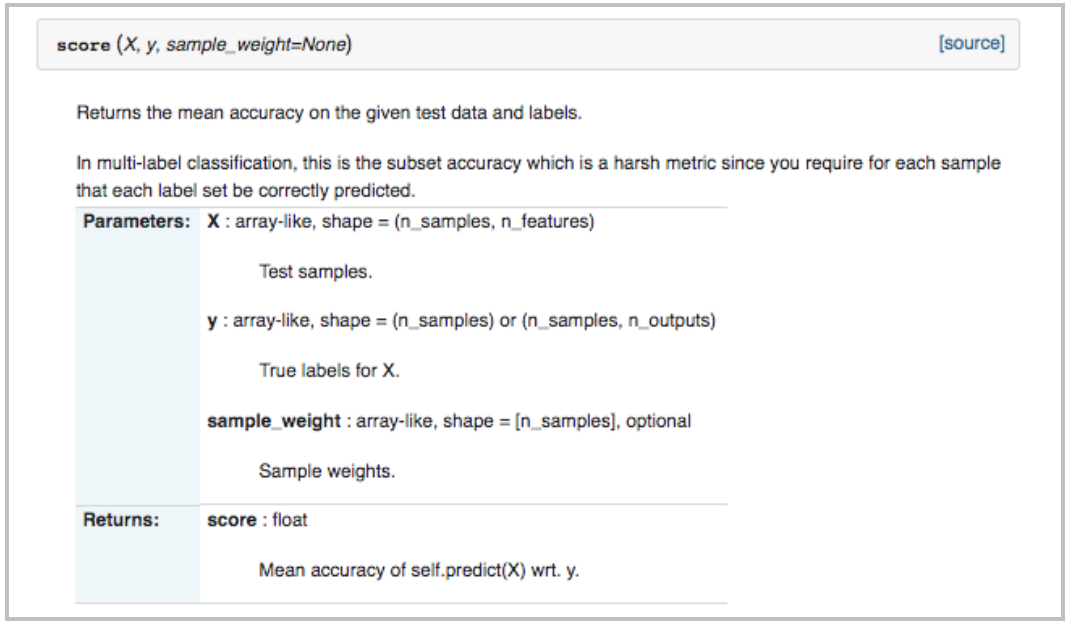
Accuracy of training model

- 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: 0.847047619047619
```

- 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.

Add final accuracy to the pickled dataframe

- Let's save our gradient boosting score in our `model_final` dataset and display it
- We can see that the gbm model did not perform as well as random forests

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

```
    metrics  values                    model
0  accuracy  0.7778                 logistic
1  accuracy  0.7778  logistic_whole_dataset
2  accuracy  0.8077          logistic_tuned
3  accuracy  0.8114           random_forest
4  accuracy  0.8072                 boosting
```

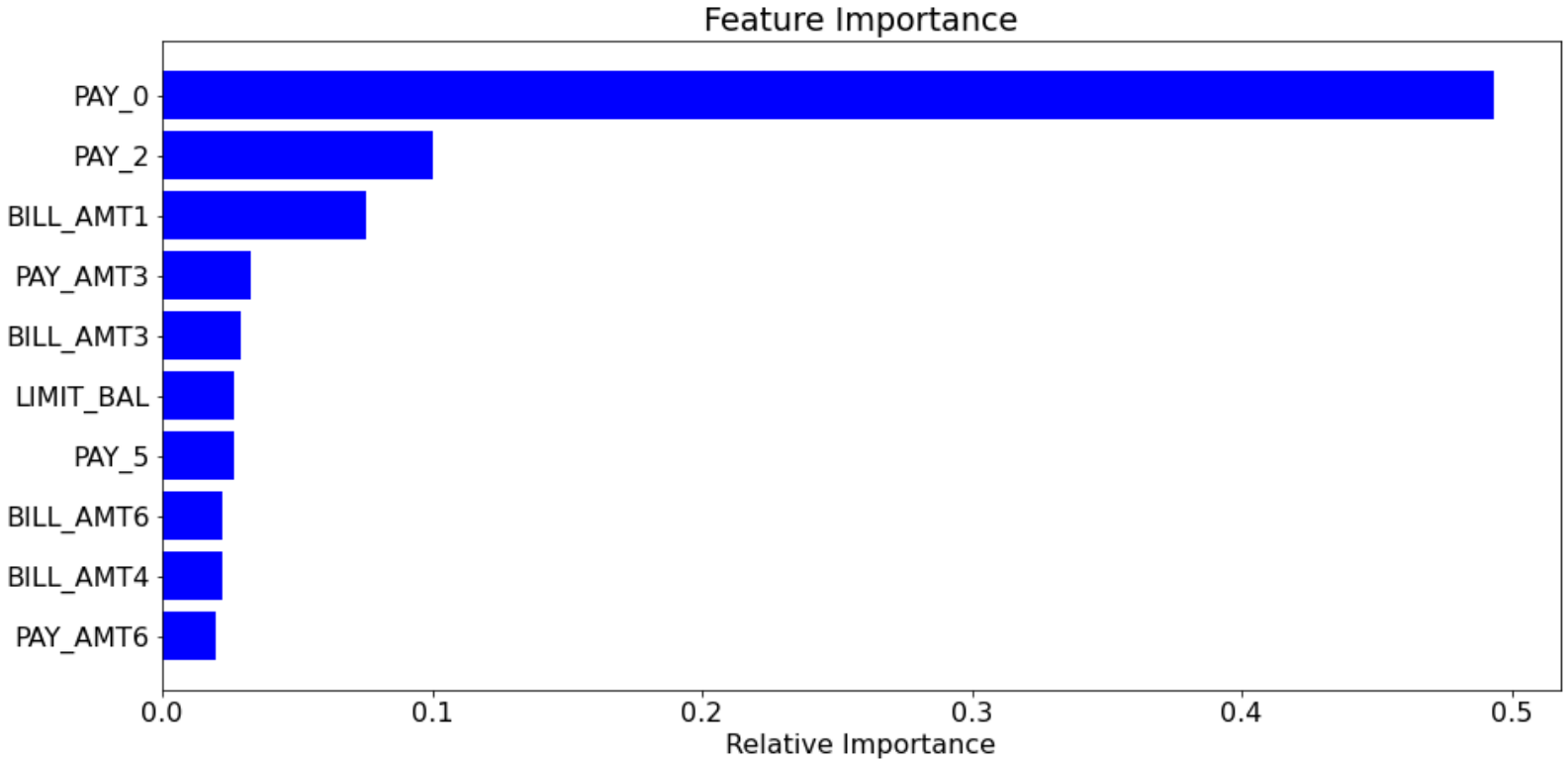
```
pickle.dump(model_final, open("model_final.sav","wb" ))
```

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

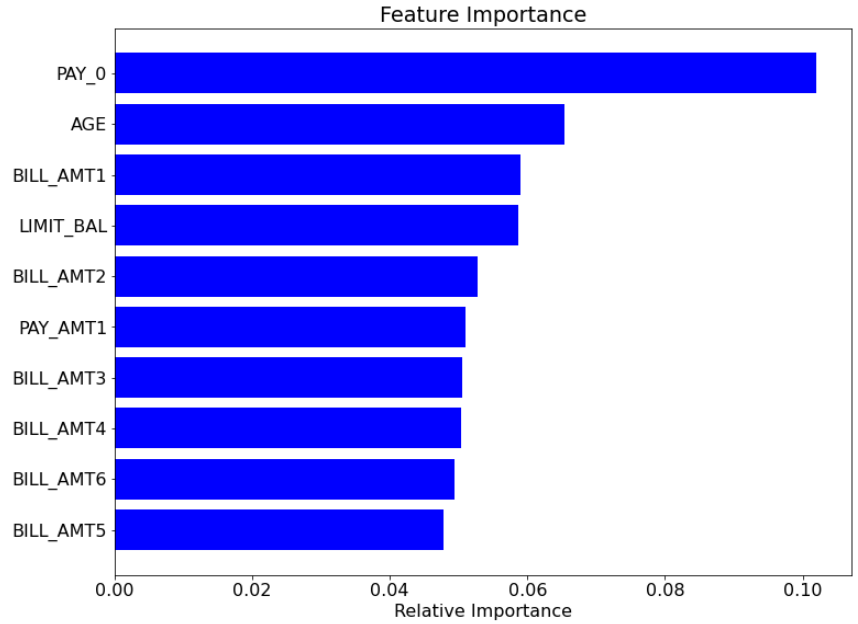
```
features = credit_card_features.columns
importances = gbm.feature_importances_
indices = np.argsort(importances)[::-1]
top_indices = indices[0:10][::-1]

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

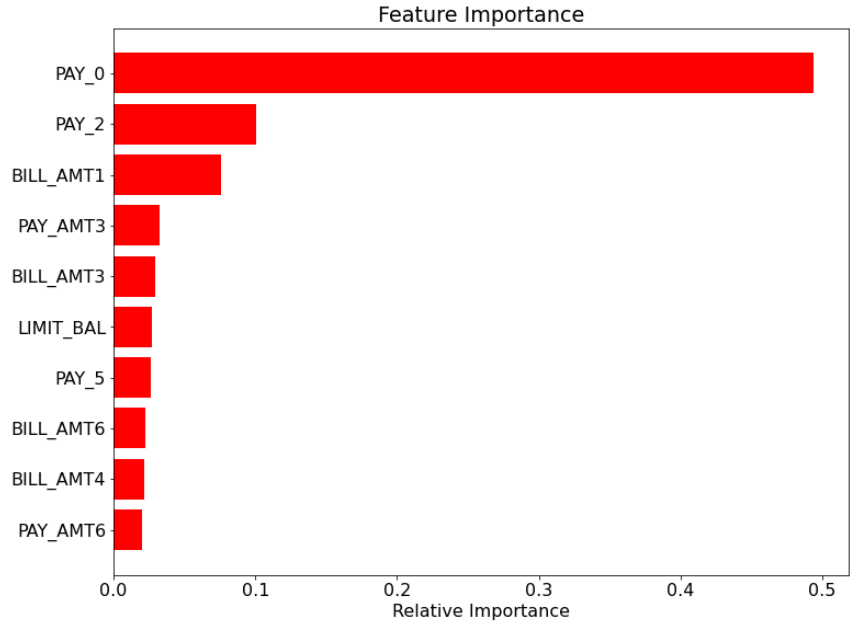


Compare feature importance plots

- **Random forests feature importance**



- **Gradient boosting feature importance**



Knowledge check 4

Exercise 3

Module completion checklist

Objective	Complete
Introduce random forests and discuss use cases	✔
Summarize the concepts associated with random forests and bagging	✔
Prepare to implement a random forest	✔
Implement random forests on the dataset	✔
Introduce gradient boosting and how it compares to bagging	✔
Discuss gradient tree boosting within scikit-learn and implement on credit card data	✔

Recap

- Earlier in the course we set a goal of using the credit card dataset to understand:

1. Which variables are the strongest predictors of default payment?
2. How does the probability of default payment vary by categories of different demographic variables?"

- **Do we have our answers?**

This completes our module

Congratulations!

Wondering what's next?

- You may be interested in the following courses, which Data Society offers for Booz Allen Hamilton:
  - **Intro to Text Mining & NLP** - This course is a foundational exploration of text-based machine learning. Students will learn cutting-edge techniques for processing, cleaning, and formatting text data and optimizing for analysis. Students will acquire skills in mining data from a corpus of structure and unstructured documents and be prepared to tackle more advanced concepts such as calculating sentiment in text data.
  - **Intro to Neural Networks** - Neural networks are a powerful tool for any data scientist to know. By the end of this course, students will understand the foundations of this complex topic and be equipped to build and optimize a neural network that can be used to model complex patterns and prediction problems.