DATA SOCIETY®

Advanced classification - day 2

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

Module completion checklist

Objective	Complete
Optimize random forest model	
Predict and evaluate the optimized model	
Optimize gradient boosting model	
Predict and evaluate the optimized boosting model	

Directory settings

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

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

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

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

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

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 for classification best practices

```
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.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.ensemble import GradientBoostingClassifier
from matplotlib.legend_handler import HandlerLine2D

# New today
from sklearn.model_selection import RandomizedSearchCV
```

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
```

Random forest vs gradient boosting

Goal for today

- Optimize a **random forest** and **gradient boosting** model to predict high yield in Costa Rican dataset
- Compare models at the end of today and decide on a **model champion**

Recap: random forest

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

Random forest methodology

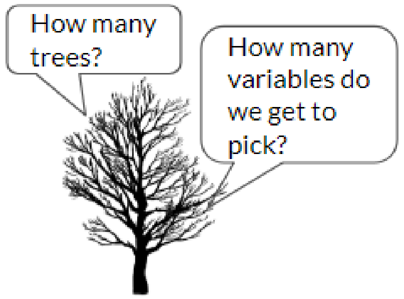
Step 1: Set *n* & *m* for forest

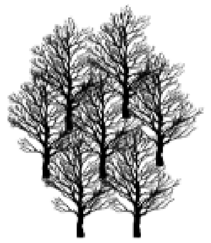
Step 2: Build forest on training data

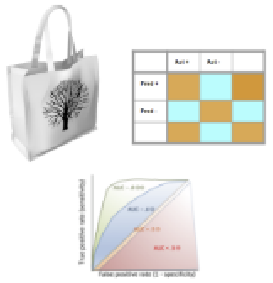
Step 3: Check performance

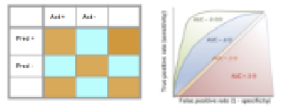
Step 4: Apply to test data & evaluate

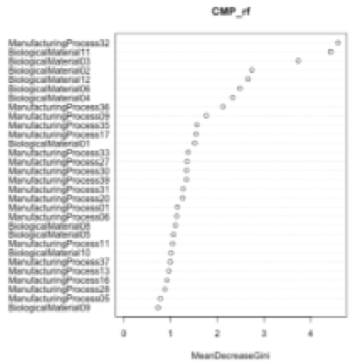
Step 5: Use variable importance as needed











Recap: why 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

Review data cleaning steps from last week

- **Today, we will be loading the cleaned dataset we used last class**
- To recap, the steps to get to this cleaned dataset were:
  - Remove household ID and individual ID
  - Remove variables with over 50% NAs
  - Transformed target variable to binary
  - Remove highly correlated variables

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

Tuning random forest model

- We saw when we built our random forest that there are many parameters
- Let's load our original model from last week
- Now we can take a look at the parameters currently in use

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

```
forest.get_params()
```

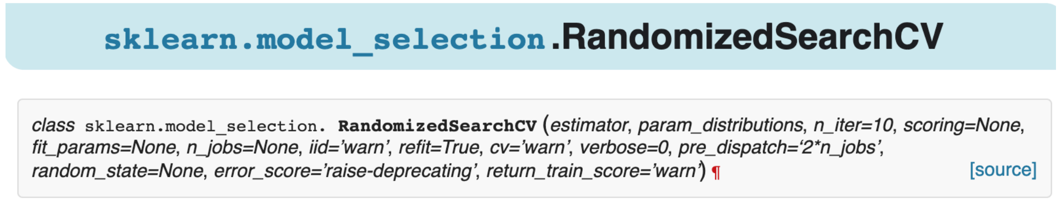
```
{'bootstrap': True, 'class_weight': None, 'criterion': 'gini', 'max_depth': None, 'max_features':
'auto', 'max_leaf_nodes': None, 'min_impurity_decrease': 0.0, 'min_impurity_split': None,
'min_samples_leaf': 1, 'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0, 'n_estimators': 100,
'n_jobs': None, 'oob_score': False, 'random_state': 1, 'verbose': 0, 'warm_start': False}
```

Tuning random forest model

- The best approach would be to narrow the range of values for each parameter
- We can use `RandomizedSearchCV` to optimize our hyperparameters to sample from during fitting
- `GridSearchCV` can be computationally expensive, especially when dealing with a large hyperparameter space
- We can search only a random subset of parameter values instead, using `RandomSearchCV`

RandomizedSearchCV

- It uses a randomized search on hyperparameters
- Unlike `GridSearchCV`, it samples a fixed number of parameter settings from specified probability distributions



sklearn.model_selection.RandomizedSearchCV

class sklearn.model_selection. **RandomizedSearchCV** (*estimator, param_distributions, n_iter=10, scoring=None, fit_params=None, n_jobs=None, iid='warn', refit=True, cv='warn', verbose=0, pre_dispatch='2*n_jobs', random_state=None, error_score='raise-deprecating', return_train_score='warn'*) ¶ [source]

Parameter grid

- Let's create a grid of parameter ranges

```
# Number of trees in random forest.
n_estimators = [int(x) for x in np.linspace(start = 200, stop = 2000, num = 10)]

# Number of features to consider at every split.
max_features = ['auto', 'sqrt']

# Maximum number of levels in tree.
max_depth = [int(x) for x in np.linspace(10, 110, num = 11)]
max_depth.append(None)

# Minimum number of samples required to split a node.
min_samples_split = [2, 5, 10]

# Minimum number of samples required at each leaf node.
min_samples_leaf = [1, 2, 4]


# Create the random grid.
random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf}
print(random_grid)
```

Set up cross-validation function

- Now we instantiate the model, using 3-fold cross-validation with 100 different combinations

```
rf_random = RandomizedSearchCV(estimator = forest,
                               param_distributions = random_grid,
                               n_iter = 100,
                               cv = 3,
                               verbose = 0,
                               random_state = 1,
                               n_jobs = -1)
# Fit the random search model.
rf_random.fit(X_train, y_train)
```

```
rf_random.best_params_
```

```
{'n_estimators': 1000,
 'min_samples_split': 5,
 'min_samples_leaf': 2,
 'max_features': 'sqrt',
 'max_depth': 10}
```

Optimized random forest model

- Now we can use these optimized hyperparameters to implement the random forest again on `X_train`

```
optimized_forest = RandomForestClassifier(criterion = 'gini',
                                          n_estimators = 1000,
                                          min_samples_split = 5,
                                          min_samples_leaf = 2,
                                          max_features = 'sqrt',
                                          max_depth = 10,
                                          random_state = 1)
```

```
optimized_forest.fit(X_train, y_train)
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=10, max_features='sqrt', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=2, min_samples_split=5,
                       min_weight_fraction_leaf=0.0, n_estimators=1000,
                       n_jobs=None, oob_score=False, random_state=1, verbose=0,
                       warm_start=False)
```

Knowledge Check 1

Exercise 1

Module completion checklist

Objective	Complete
Optimize random forest model	✔
Predict and evaluate the optimized model	
Optimize gradient boosting model	
Predict and evaluate the optimized boosting model	

Predict using the best model parameters

- Prediction class for each observation as a list

```
optimized_forest_y_predict = optimized_forest.predict(X_test)
# Look at the first few predictions.
print(optimized_forest_y_predict[0:5, ])
```

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

```
optimized_forest_accuracy = metrics.accuracy_score(y_test, optimized_forest_y_predict)
print ("Accuracy on test data (best model): ", optimized_forest_accuracy)
```

```
Accuracy on test data (best model):  0.8535564853556485
```

Optimized random forest: 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_optimized =
pickle.load(open("model_final_forest_gbm.sav","rb"))
```

Optimized random forest: save final accuracy

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

```
    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
11  accuracy  0.8536            optimized forest
```

Knowledge Check 2

Exercise 2

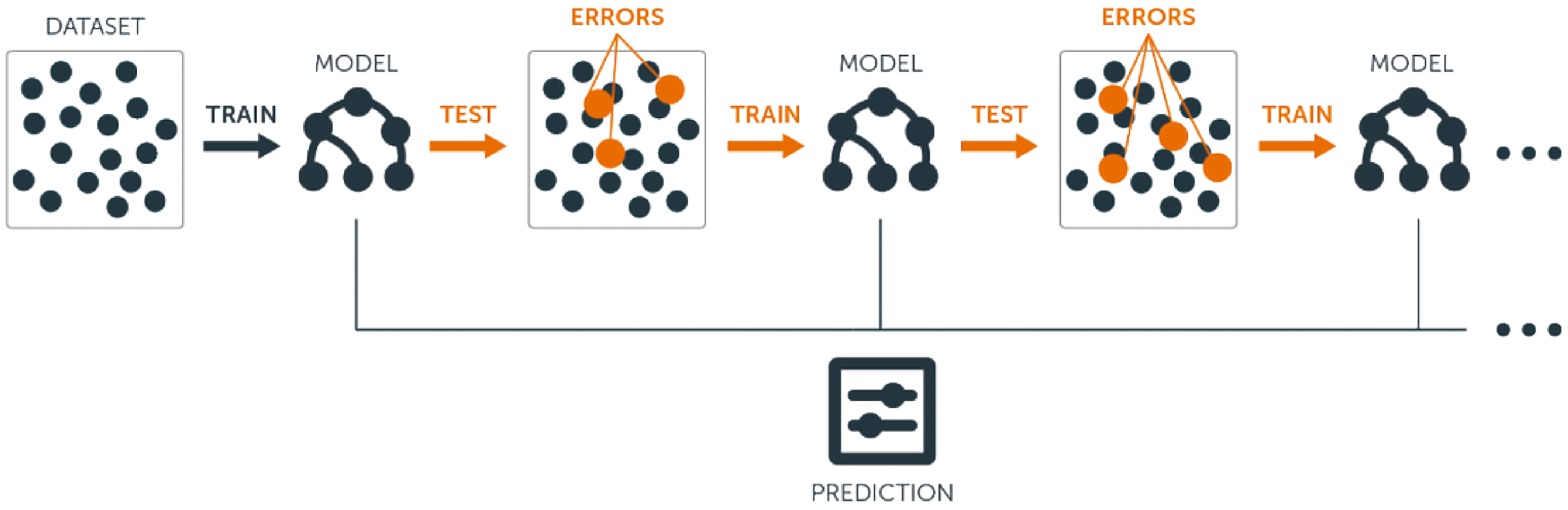
Module completion checklist

Objective	Complete
Optimize random forest model	✔
Predict and evaluate the optimized model	✔
Optimize gradient boosting model	
Predict and evaluate the optimized boosting model	

Recap: boosting intuition

- 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:
  - it sees the **errors from a decision tree** on the dataset
  - it **identifies the pattern of the errors and builds a new decision tree** on them
  - it **repetitively leverages these patterns in residuals to strengthen the overall model**

Gradient boosted trees



Source

Recap: 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 MSE
  - **Binary classification**: ***logistic function***

Ways to optimize gradient boosting

- We saw when we built our boosting model that there are many parameters
- All the values of our original boosting model are set to the `GradientBoostingClassifier` defaults within `sklearn`
- We are now going to optimize the model **focusing on the four parameters we called out**
  - `max_depth = None`
  - `min_samples_split = 2`
  - `min_samples_leaf = 1`
  - `max_features = None`

Define an optimal number function

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

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

Optimize: learning rate

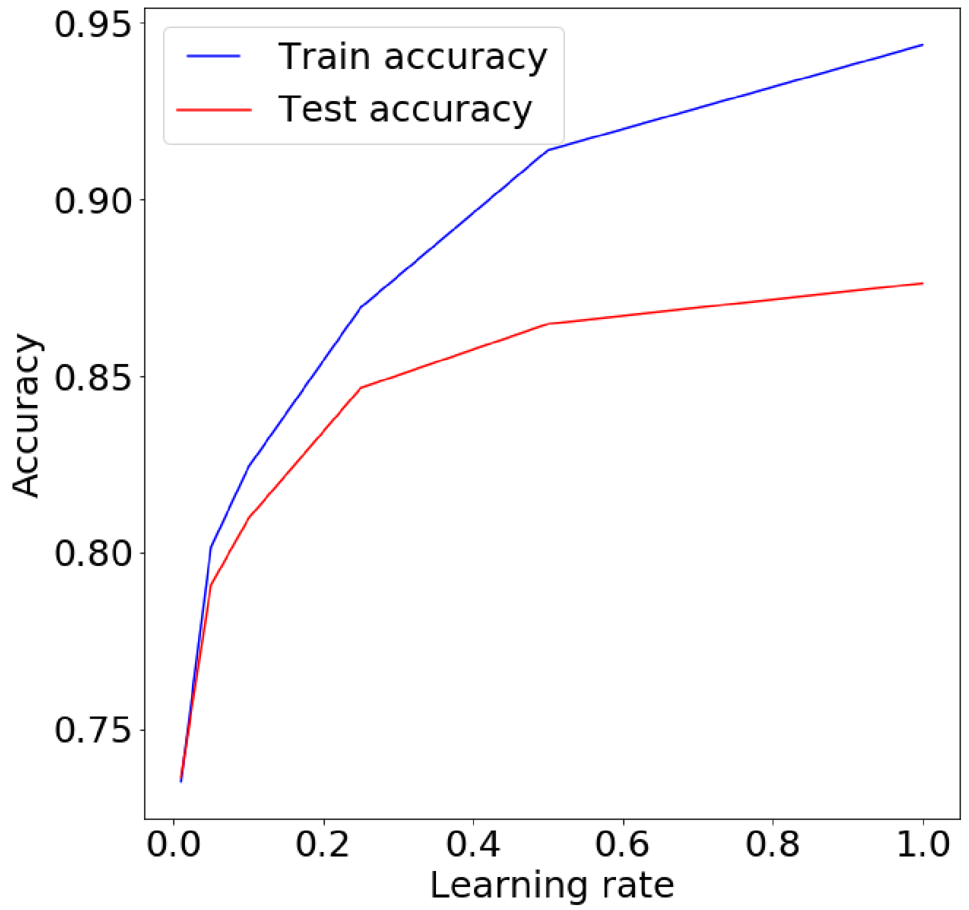
```
# Learning Rate
learning_rates = [1, 0.5, 0.25, 0.1, 0.05, 0.01]
train_results = []
test_results = []


for eta in learning_rates:
   model = GradientBoostingClassifier(learning_rate=eta)
   model.fit(X_train, y_train)
   train_pred = model.predict(X_train)
   acc_train = accuracy_score(y_train, train_pred)
   train_results.append(acc_train)
   y_pred = model.predict(X_test)
   acc_test = accuracy_score(y_test, y_pred)
   test_results.append(acc_test)
```

```
optimal_learning_rate = optimal_parameter(learning_rates, test_results)
```

Plot: learning rate

```
from matplotlib.legend_handler import
HandlerLine2D
line1, = plt.plot(learning_rates,
train_results,'b', label= "Train accuracy")
line2, = plt.plot(learning_rates, test_results,
'r', label= "Test accuracy")
plt.legend(handler_map={line1:
HandlerLine2D(numpoints=2)})
plt.ylabel("Accuracy")
plt.xlabel("Learning rate")
plt.show()
```



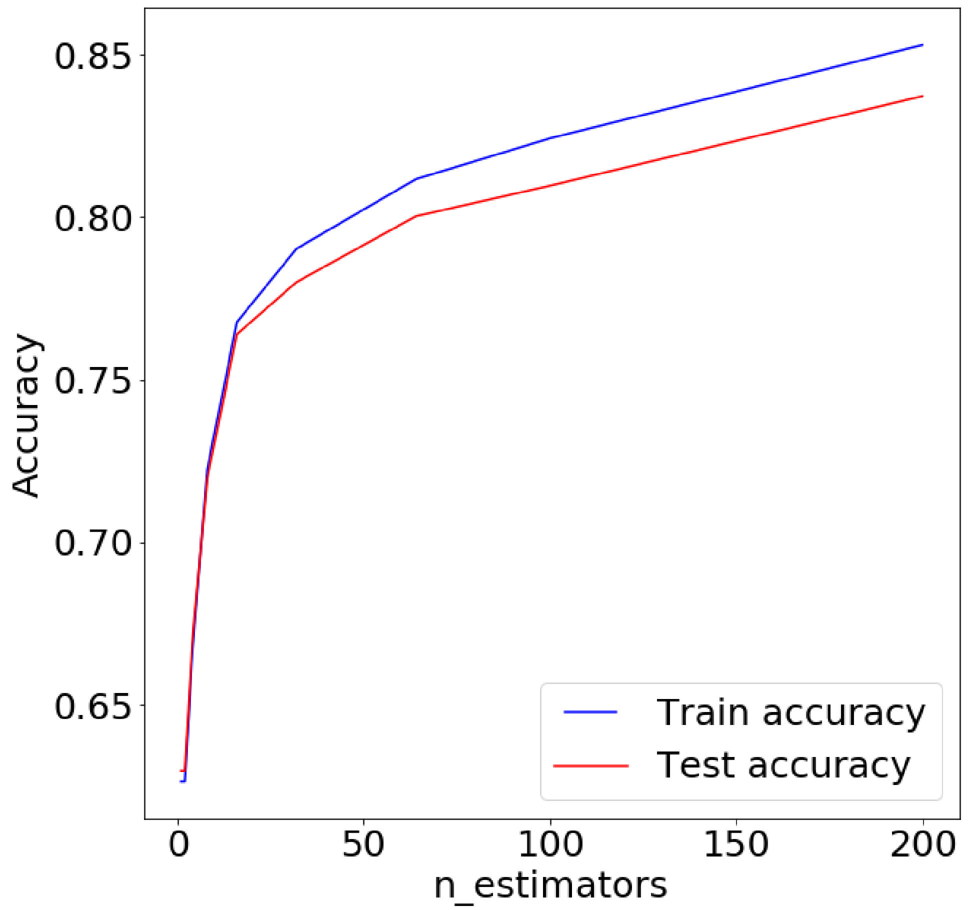
Optimize: n estimators

```
n_estimators = [1, 2, 4, 8, 16, 32, 64, 100, 200]
train_results = []
test_results = []
for estimator in n_estimators:
   model = GradientBoostingClassifier(n_estimators=estimator)
   model.fit(X_train, y_train)
   train_pred = model.predict(X_train)
   acc_train = accuracy_score(y_train, train_pred)
   train_results.append(acc_train)
   y_pred = model.predict(X_test)
   acc_test = accuracy_score(y_test, y_pred)
   test_results.append(acc_test)
```

```
optimal_n_estimators = optimal_parameter(n_estimators, test_results)
```

Plot: n estimators

```
from matplotlib.legend_handler import
HandlerLine2D
line1, = plt.plot(n_estimators, train_results,
'b', label= "Train accuracy")
line2, = plt.plot(n_estimators, test_results,
'r', label= "Test accuracy")
plt.legend(handler_map={line1:
HandlerLine2D(numpoints=2)})
plt.ylabel('Accuracy')
plt.xlabel('n_estimators')
plt.show()
```



Optimize: max depth

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

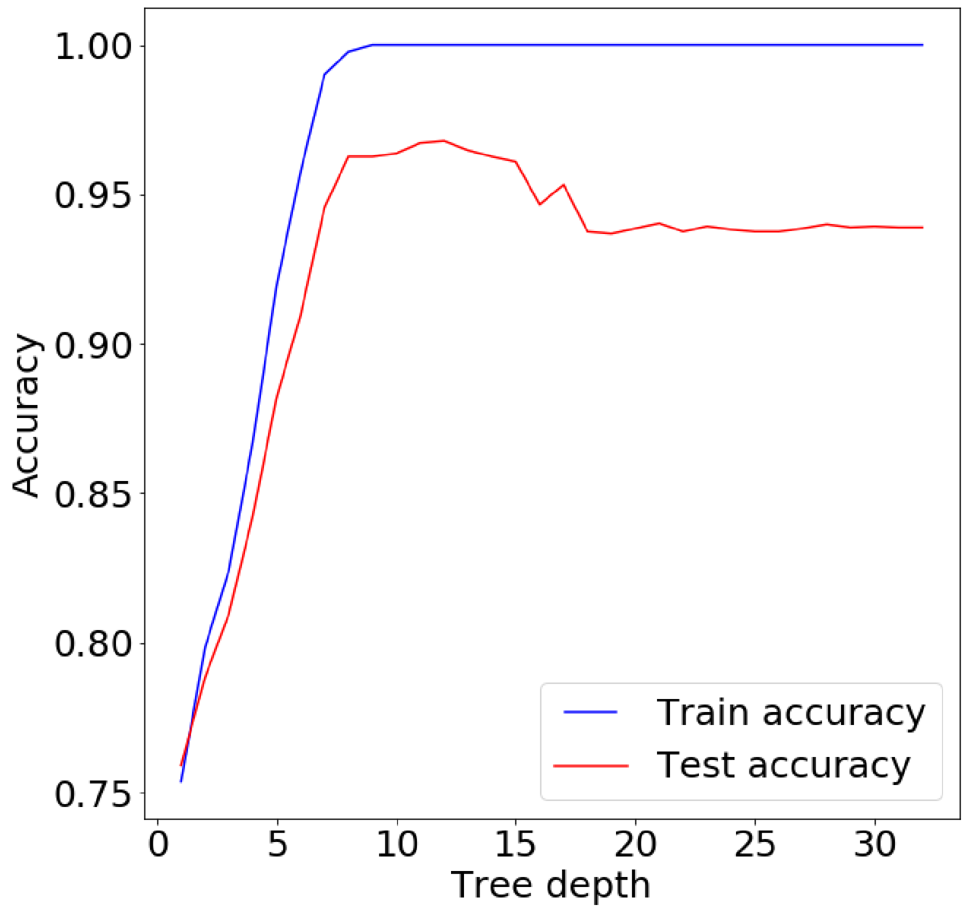
```
# Max depth:
max_depths = np.linspace(1, 32, 32, endpoint=True)
train_results = []
test_results = []
for max_depth in max_depths:
   model = GradientBoostingClassifier(max_depth=max_depth)
   model.fit(X_train, y_train)
   train_pred = model.predict(X_train)
   acc_train = accuracy_score(y_train, train_pred)
   train_results.append(acc_train)
   y_pred = model.predict(X_test)
   acc_test = accuracy_score(y_test, y_pred)
   test_results.append(acc_test)
```

```
# Store optimal max_depth.
optimal_max_depth = optimal_parameter(max_depths, test_results)
```

Plot: max depth

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

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



Optimize: min samples split

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

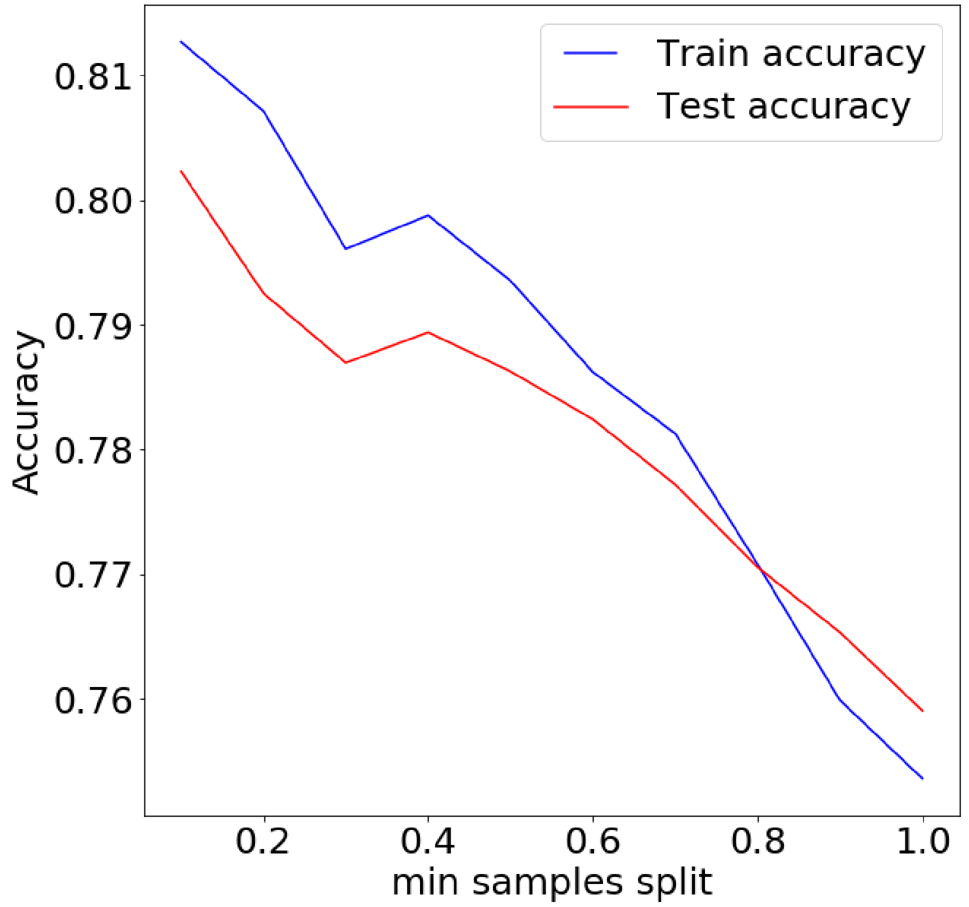
```
min_samples_splits = np.linspace(0.1, 1.0, 10, endpoint = True)
train_results = []
test_results = []
for min_samples_split in min_samples_splits:
   model = GradientBoostingClassifier(min_samples_split = min_samples_split)
   model.fit(X_train, y_train)
   train_pred = model.predict(X_train)
   acc_train = accuracy_score(y_train, train_pred)
   train_results.append(acc_train)
   y_pred = model.predict(X_test)
   acc_test = accuracy_score(y_test, y_pred)
   test_results.append(acc_test)
```

```
# Store optimal min_samples_split.
optimal_min_samples_split = optimal_parameter(min_samples_splits, test_results)
```

Plot: min samples split

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

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



Optimize: min samples leaf

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

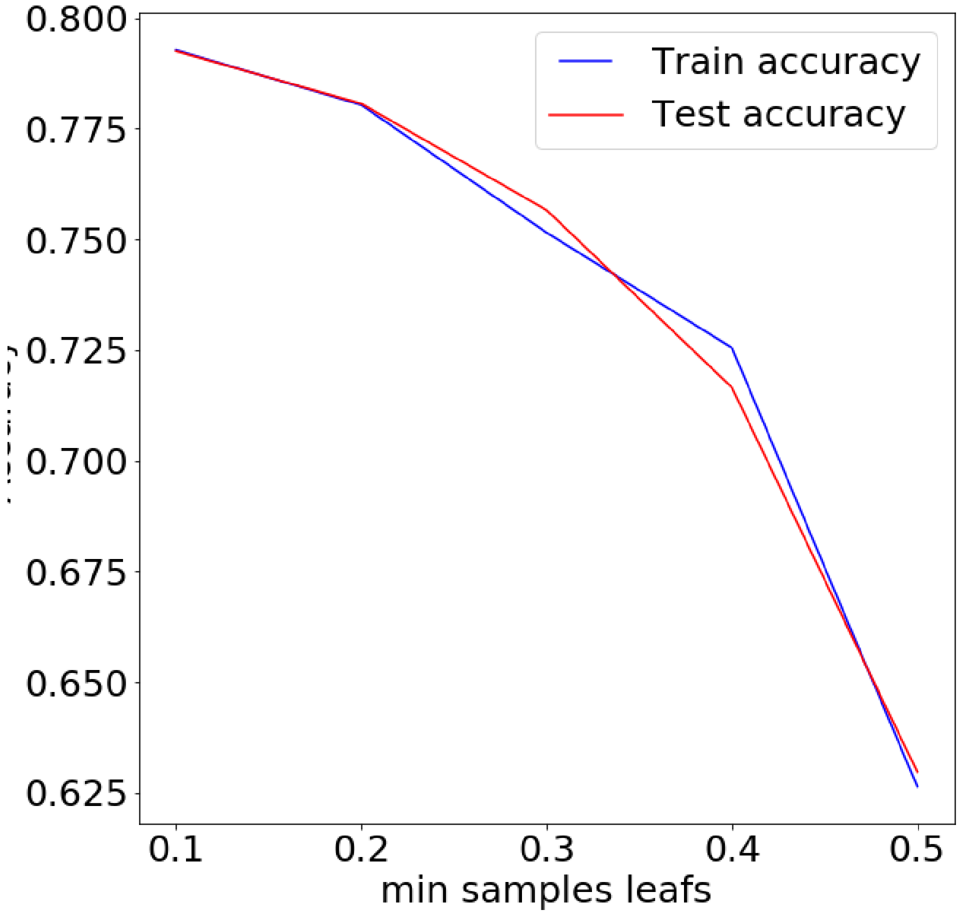
```
# Min_samples_leaf:
min_samples_leafs = np.linspace(0.1, 0.5, 5, endpoint = True)
train_results = []
test_results = []
for min_samples_leaf in min_samples_leafs:
   model = GradientBoostingClassifier(min_samples_leaf = min_samples_leaf)
   model.fit(X_train, y_train)
   train_pred = model.predict(X_train)
   acc_train = accuracy_score(y_train, train_pred)
   train_results.append(acc_train)
   y_pred = model.predict(X_test)
   acc_test = accuracy_score(y_test, y_pred)
   test_results.append(acc_test)
```

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

Plot: min samples leaf

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

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



Optimize: max features

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

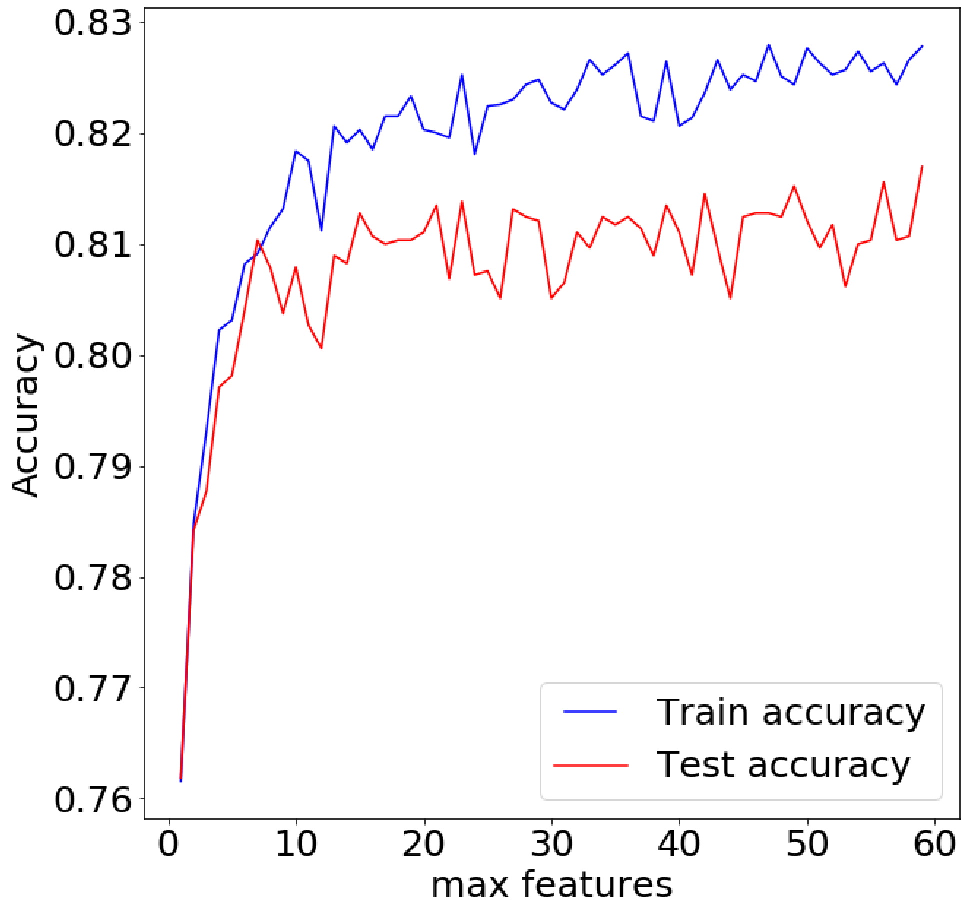
```
# Max_features:
max_features = list(range(1,X.shape[1]))
train_results = []
test_results = []
for max_feature in max_features:
  model = GradientBoostingClassifier(max_features = max_feature)
  model.fit(X_train, y_train)
  train_pred = model.predict(X_train)
  acc_train = accuracy_score(y_train,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

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

Plot: max features

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

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



Optimized model

- Now that we have now walked through four parameters that will help us optimize our gradient boosting model
- Let's look at what each of the optimal parameters are:

```
print("The optimal learning rate is:",
optimal_learning_rate)
```

```
The optimal learning rate is: 1
```

```
print("The optimal number of estimators is:",
optimal_n_estimators)
```

```
The optimal number of estimators is: 200
```

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

```
The optimal max depth is: 3.0
```

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

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

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

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

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

```
The optimal max features is: 59
```

Knowledge Check 3

Exercise 3

Module completion checklist

Objective	Complete
Optimize random forest model	✔
Predict and evaluate the optimized model	✔
Optimize gradient boosting model	✔
Predict and evaluate the optimized boosting model	

Build optimized model

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

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

# Implement the decision tree on X_train.
gbm_optimized = GradientBoostingClassifier(learning_rate = optimal_learning_rate,
                                           n_estimators = optimal_n_estimators,
                                           max_depth = optimal_max_depth,
                                           min_samples_split = optimal_min_samples_split,
                                           min_samples_leaf = optimal_min_samples_leafs,
                                           max_features = optimal_max_features)

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

```
GradientBoostingClassifier(criterion='friedman_mse', init=None, learning_rate=1,
                           loss='deviance', max_depth=3.0, max_features=59,
                           max_leaf_nodes=None, min_impurity_decrease=0.0,
                           min_impurity_split=None, min_samples_leaf=0.1,
                           min_samples_split=0.1, min_weight_fraction_leaf=0.0,
                           n_estimators=200, n_iter_no_change=None,
                           presort='auto', random_state=None, subsample=1.0,
                           tol=0.0001, validation_fraction=0.1, verbose=0,
                           warm_start=False)
```

```
gbm_optimized_fit = gbm_optimized.fit(X_train, y_train)
```

Predict with optimized model

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

```
# Predict on X_test.
y_predict_gbm_optimized = gbm_optimized.predict(X_test)

# Accuracy score.
acc_score_gbm_optimized = accuracy_score(y_test, y_predict_gbm_optimized)

print(acc_score_gbm_optimized)
```

```
0.8563458856345886
```

Add accuracy score to the final scores

- So we have it, let's add this score to the dataframe `model_final` that we created earlier
- Let's append the score to `model_final_optimized`

```
model_final_optimized = model_final_optimized.append({'metrics' : "accuracy" ,
                                                      'values' : round(acc_score_gbm_optimized,4),
                                                      'model':'gbm_optimized'} ,
                                                      ignore_index = True)
print(model_final_optimized)
```

```
     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
11  accuracy  0.8536            optimized forest
12  accuracy  0.8563               gbm_optimized
```

- Now that we have built all our models and have our final accuracy, let's discuss our results

Discuss model champion

- When we look for the best model, we do not just want to take the model with the highest accuracy score
- Remember to consider:
  - bias in models
  - optimized vs. biased models
- **We now want to open the class up for discussion about which model you would choose, and why**

Final scores

```
print(model_final_optimized)
```

```
     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
11  accuracy  0.8536               optimized forest
12  accuracy  0.8563                  gbm_optimized
```

```
pickle.dump(model_final_optimized, open("model_final_optimized_ensemble.sav", "wb" ))
```

Knowledge Check 4

Exercise 4

Module completion checklist

Objective	Complete
Optimize random forest model	✔
Predict and evaluate the optimized model	✔
Optimize gradient boosting model	✔
Predict and evaluate the optimized boosting model	✔

Workshop: next steps!

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

  - Step 1: Load data, clean data to get ready for exploratory data analysis
  - Step 2: Look for patterns by using visualizations and also unsupervised learning
  - Step 3: Using any insights gained from steps 1 and 2, move forward with building a base data table for modeling
  - Step 4: Determine your target variable, this will be the variable of interest, what you want to predict
  - Step 5: Build classification models using class materials as guidance, save accuracy scores like we did with out `model_final` dataframes
  - Step 6: Determine a model champion

This completes our module

Congratulations!