# Machine Learning with Tree-Based Models

Fundamental concepts in supervised machine learning

Black Raven (James Ng)
Aug 10 · 33 min read ★

This is a memo to share what I have learnt in Machine Learning with Tree-Based Models (using Python), capturing the learning objectives as well as my personal notes. The course is taught by Elie Kawerk from DataCamp, and it includes 5 chapters:

Chapter 1. Classification and Regression Trees
Chapter 2. The Bias-Variance Tradeoff
Chapter 3. Bagging and Random Forests
Chapter 4. Boosting
Chapter 5. Model Tuning

Photo by Jens Lelie on Unsplash

Decision trees are supervised learning models used for problems involving classification and regression. Tree models present a high flexibility that comes at a price: on one hand, trees are able to capture complex non-linear relationships; on the other hand, they are prone to memorizing the noise present in a dataset.

By aggregating the predictions of trees that are trained differently, ensemble methods take advantage of the flexibility of trees while reducing their tendency to memorize noise. Ensemble methods are used across a variety of fields and have a proven track record of winning many machine learning competitions.

In this course, you'll learn how to use Python to train decision trees and tree-based models with the user-friendly scikit-learn machine learning library. You'll understand the advantages and shortcomings of trees and demonstrate how ensembling can alleviate these shortcomings, all while practicing on real-world datasets. Finally, you'll also understand how to tune the most influential hyperparameters in order to get the most out of your models.

. . .

## Chapter 1. Classification and Regression Trees

Classification and Regression Trees (CART) are a set of supervised learning models used for problems involving classification and regression. In this chapter, you'll be introduced to the CART algorithm.

### Decision tree for classification

Classification tree is a sequence of if-else questions about individual features, in order to infer the final class labels.
· able to capture non-linear relationships between features and labels
· do not require feature scaling (e.g. standardization, min-max scaling, etc)

```
# Import DecisionTreeClassifier
```

```
from sklearn.tree import DecisionTreeClassifier
# Import train_test_split
from sklearn.model_selection import train_test_split
# Import accuracy_score
from sklearn.metrics import accuracy_score

# Split dataset into 80% train, 20% test
X_train, X_test, y_train, y_test= train_test_split(X, y,
                                                   test_size=0.2,
                                                   stratify=y,
                                                   random_state=1)

# Instantiate dt
dt = DecisionTreeClassifier(max_depth=2, random_state=1)

# Fit dt to the training set
dt.fit(X_train,y_train)

# Predict test set labels
y_pred = dt.predict(X_test)

# Evaluate test-set accuracy
accuracy_score(y_test, y_pred)
```

```
0.903508771929824 59
```

**Decision Boundary** of a Linear Model is a straight line.
**Decision Regions** produced by CART model are rectangular.

### Train your first classification tree

In this exercise you'll work with the **Wisconsin Breast Cancer Dataset** from the UCI machine learning repository. You'll predict whether a tumor is malignant or benign based on two features: the mean radius of the tumor ( `radius_mean` ) and its mean number of concave points ( `concave points_mean` ).

The dataset is already loaded in your workspace and is split into 80% train and 20% test. The feature matrices are assigned to `X_train` and `X_test` , while the arrays of labels are

assigned to `y_train` and `y_test` where class 1 corresponds to a malignant tumor and class 0 corresponds to a benign tumor. To obtain reproducible results, we also defined a variable called `SEED` which is set to 1.

```
# Import DecisionTreeClassifier from sklearn.tree
from sklearn.tree import DecisionTreeClassifier

# Instantiate a DecisionTreeClassifier with a max depth of 6
dt = DecisionTreeClassifier(max_depth=6, random_state=SEED)

# Fit dt to the training set
dt.fit(X_train, y_train)

# Predict test set labels
y_pred = dt.predict(X_test)
print(y_pred[0:5])
```

```
<script.py> output:
    [0 0 0 1 0]
```

You can see the first five predictions made by the fitted tree on the test set

## Evaluate the classification tree

Now that you've fit your first classification tree, it's time to evaluate its performance on the test set. You'll do so using the accuracy metric which corresponds to the fraction of correct predictions made on the test set.

The trained model `dt` from the previous exercise is loaded in your workspace along with the test set features matrix `X_test` and the array of labels `y_test`.

```
# Import accuracy_score
from sklearn.metrics import accuracy_score

# Predict test set labels
y_pred = dt.predict(X_test)

# Compute test set accuracy
acc = accuracy_score(y_test, y_pred)
print("Test set accuracy: {:.2f}".format(acc))
```

```
<script.py> output:
    Test set accuracy: 0.89
```

Using only two features, your tree was able to achieve an accuracy of 89%!

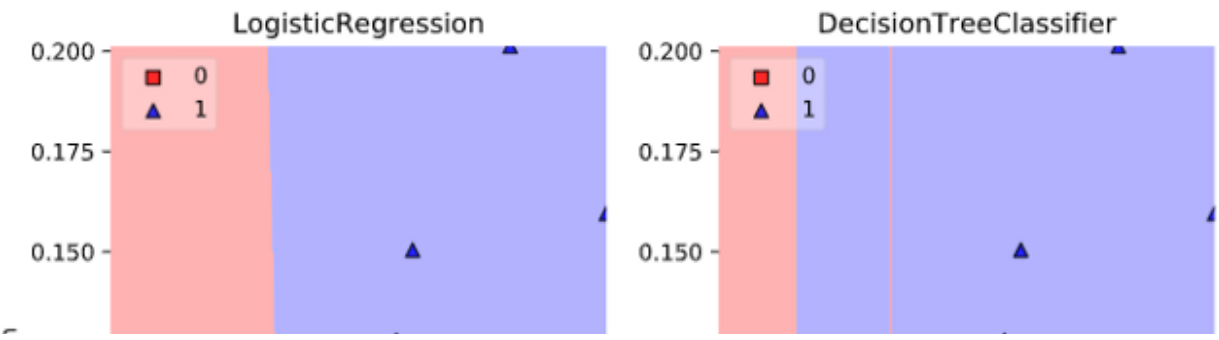
## Logistic regression vs classification tree

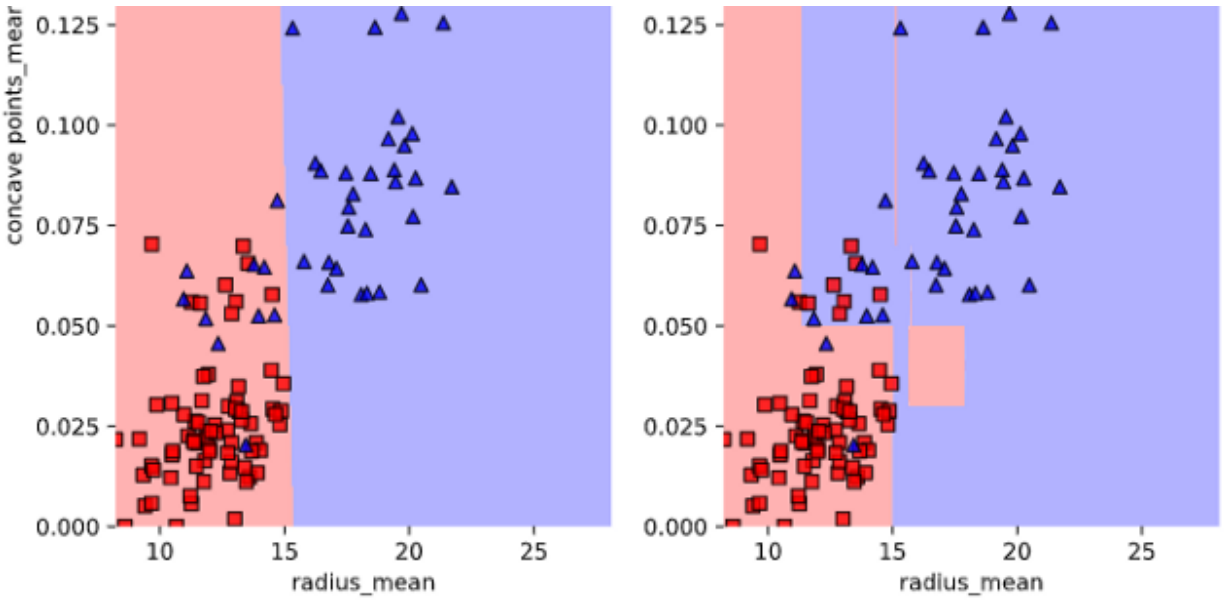
A classification tree divides the feature space into **rectangular regions**. In contrast, a linear model such as logistic regression produces only a single linear decision boundary dividing the feature space into two decision regions.

We have written a custom function called `plot_labeled_decision_regions()` that you can use to plot the decision regions of a list containing two trained classifiers. You can type `help(plot_labeled_decision_regions)` in the IPython shell to learn more about this function.

`X_train`, `X_test`, `y_train`, `y_test`, the model `dt` that you've trained previously, as well as the function `plot_labeled_decision_regions()` are available in your workspace.

```
# Import LogisticRegression from sklearn.linear_model
from sklearn.linear_model import LogisticRegression

# Instatiate logreg
logreg = LogisticRegression(random_state=1)

# Fit logreg to the training set
logreg.fit(X_train, y_train)

# Define a list containing the two classifiers logreg and dt
clfs = [logreg, dt]

# Review the decision regions of the two classifiers
plot_labeled_decision_regions(X_test, y_test, clfs)
```

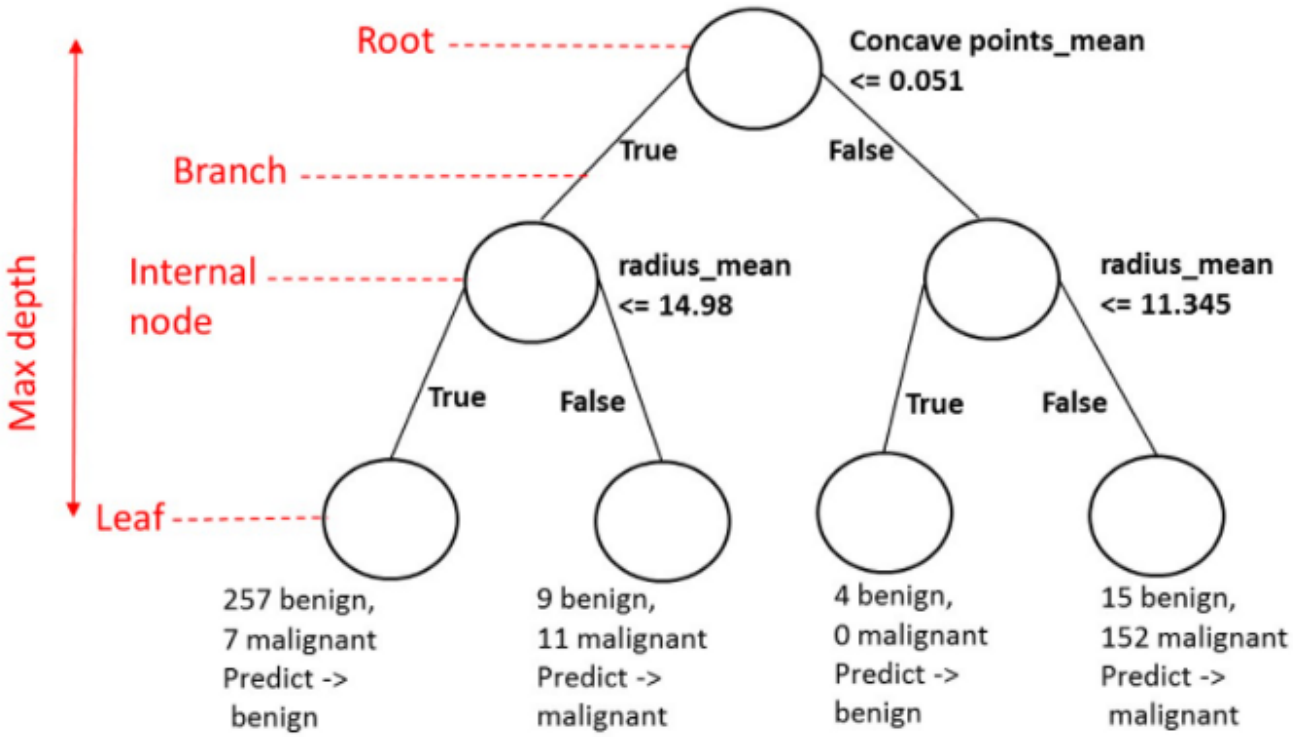





Notice how the decision boundary produced by logistic regression is linear while the boundaries produced by the classification tree divide the feature space into rectangular regions.

## Classification tree Learning

Decision-Tree has data structure consisting of a hierarchy of nodes. Each node is a point that involves either a question or a prediction.




## Growing a classification tree

The growth of an unconstrained classification tree followed a few simple rules. Which of the following is **not** one of these rules?

- The existence of a node depends on the state of its predecessors.
- The impurity of a node can be determined using different criteria such as entropy and the gini-index.
- When the information gain resulting from splitting a node is null, the node is declared as a leaf.
- When an internal node is split, the split is performed in such a way so that information gain is minimized.

Answer: When an internal node is split, the split is performed in such a way so that information gain (IG) is minimized. Actually, splitting an internal node always involves **maximizing IG**!

### Using entropy as a criterion

In this exercise, you'll train a classification tree on the Wisconsin Breast Cancer dataset using entropy as an information criterion. You'll do so using all the 30 features in the dataset, which is split into 80% train ( `X_train` & `y_train` ) and 20% test.

```
# Import DecisionTreeClassifier from sklearn.tree
from sklearn.tree import DecisionTreeClassifier

# Instantiate dt_entropy, set 'entropy' as the information criterion
dt_entropy = DecisionTreeClassifier(max_depth=8, criterion='entropy',
random_state=1)

# Fit dt_entropy to the training set
dt_entropy.fit(X_train, y_train)
```

### Entropy vs Gini index

In this exercise you'll compare the test set accuracy of `dt_entropy` to the accuracy of another tree named `dt_gini` . The tree `dt_gini` was trained on the same dataset using the same parameters except for the information criterion which was set to the gini index using the keyword `'gini'` .

`X_test`, `y_test`, `dt_entropy`, as well as `accuracy_gini` which corresponds to the test set accuracy achieved by `dt_gini` are available in your workspace.

```
# Import accuracy_score from sklearn.metrics
from sklearn.metrics import accuracy_score

# Use dt_entropy to predict test set labels
y_pred= dt_entropy.predict(X_test)

# Evaluate accuracy_entropy
accuracy_entropy = accuracy_score(y_test, y_pred)

# Print accuracy_entropy
print('Accuracy achieved by using entropy:', accuracy_entropy)

# Print accuracy_gini
print('Accuracy achieved by using the gini index:', accuracy_gini)
```

```
<script.py> output:
    Accuracy achieved by using entropy: 0.929824561404
    Accuracy achieved by using the gini index: 0.929824561404
```

Notice how the two models achieve exactly the same accuracy. Most of the time, the gini index and entropy lead to the same results. The **gini index** is slightly faster to compute and is the **default** criterion used in the `DecisionTreeClassifier` model of scikit-learn.

## Decision tree for regression

In regression, the target variable is continuous, so the model output is a real value.

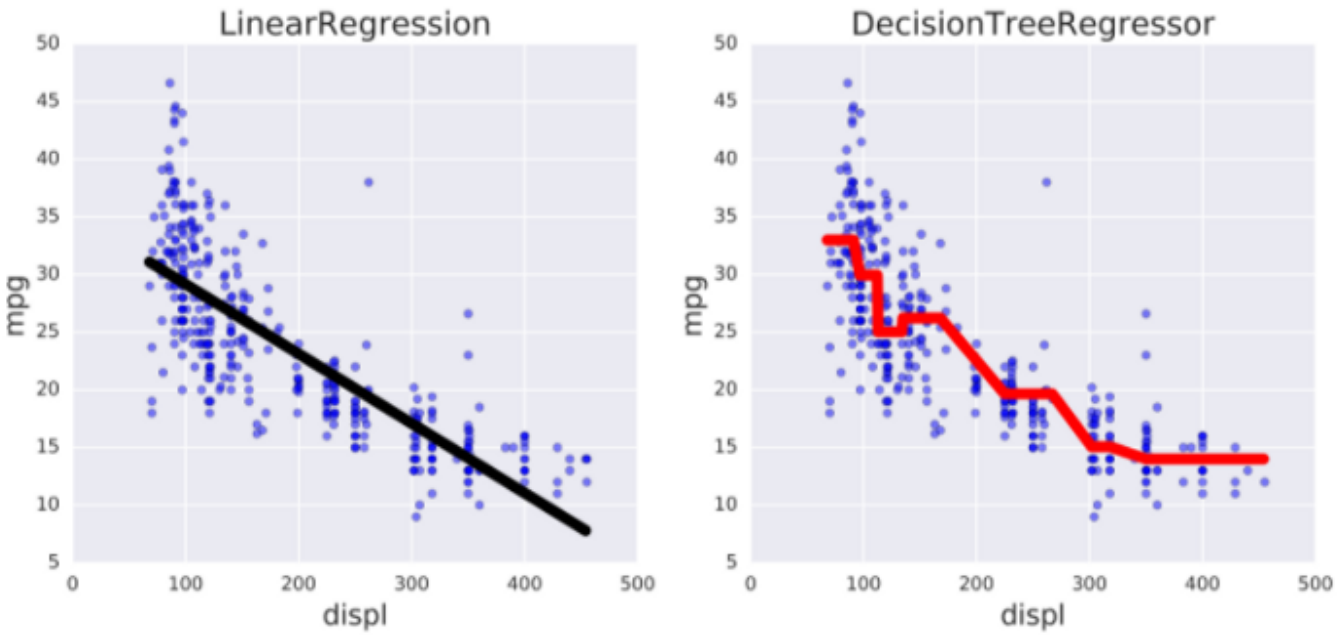
In a non-linear regression problem, Decision-Tree is able to capture the non-linear relationship between feature and target.

```
# Import DecisionTreeRegressor
from sklearn.tree import DecisionTreeRegressor
# Import train_test_split
from sklearn.model_selection import train_test_split
# Import mean_squared_error as MSE
from sklearn.metrics import mean_squared_error as MSE

# Split data into 80% train and 20% test
```

```
# Split data into 80% train and 20% test
X_train, X_test, y_train, y_test= train_test_split(X, y,
                                                   test_size=0.2,
                                                   random_state=3)

# Instantiate a DecisionTreeRegressor 'dt'
dt = DecisionTreeRegressor(max_depth=4,
                           min_samples_leaf=0.1,
                           random_state=3)

# Fit 'dt' to the training-set
dt.fit(X_train, y_train)
# Predict test-set labels
y_pred = dt.predict(X_test)

# Compute test-set MSE
mse_dt =  MSE(y_test, y_pred)

# Compute test-set RMSE
rmse_dt = mse_dt**(1/2)

# Print rmse_dt
print(rmse_dt)
```

to impose a stopping condition in which each leaf has to contain at least 10% of the training data

```
5.1023068889
```




miles per gallon vs displacement (Auto-mpg Dataset)

### Train your first regression tree

In this exercise, you'll train a regression tree to predict the `mpg` (miles per gallon) consumption of cars in the **auto-mpg dataset** using all the six available features.

The dataset has been split to 80% train ( `X_train` & `y_train` ) and 20% test.

```
# Import DecisionTreeRegressor from sklearn.tree
from sklearn.tree import DecisionTreeRegressor

# Instantiate dt
dt = DecisionTreeRegressor(max_depth=8,
             min_samples_leaf=0.13,
            random_state=3)

# Fit dt to the training set
dt.fit(X_train, y_train)
```

### Evaluate the regression tree

In this exercise, you will evaluate the test set performance of `dt` using the Root Mean Squared Error (RMSE) metric. The RMSE of a model measures, on average, how much the model's predictions differ from the actual labels. The RMSE of a model can be obtained by computing the square root of the model's Mean Squared Error (MSE).

The features matrix `X_test` , the array `y_test` , as well as the decision tree regressor `dt` that you trained in the previous exercise are available in your workspace.

```
# Import mean_squared_error from sklearn.metrics as MSE
from sklearn.metrics import mean_squared_error as MSE

# Compute y_pred
y_pred = dt.predict(X_test)

# Compute mse_dt
mse_dt = MSE(y_test, y_pred)

# Compute rmse_dt
rmse_dt = mse_dt**0.5

# Print rmse_dt
print("Test set RMSE of dt: {:.2f}".format(rmse_dt))
```

```
<script.py> output:
    Test set RMSE of dt: 4.37
```

### Linear regression vs regression tree

In this exercise, you'll compare the test set RMSE of `dt` to that achieved by a linear regression model. We have already instantiated a linear regression model `lr` and trained it on the same dataset as `dt`.

The features matrix `X_test`, the array of labels `y_test`, the trained linear regression model `lr`, `mean_squared_error` function which was imported under the alias `MSE` and `rmse_dt` from the previous exercise are available in your workspace.

```
# Predict test set labels
y_pred_lr = lr.predict(X_test)

# Compute mse_lr
mse_lr = MSE(y_test, y_pred_lr)

# Compute rmse_lr
rmse_lr = mse_lr**0.5

# Print rmse_lr
print('Linear Regression test set RMSE: {:.2f}'.format(rmse_lr))

# Print rmse_dt
print('Regression Tree test set RMSE: {:.2f}'.format(rmse_dt))
```

```
<script.py> output:
    Linear Regression test set RMSE: 5.10
    Regression Tree test set RMSE: 4.37
```

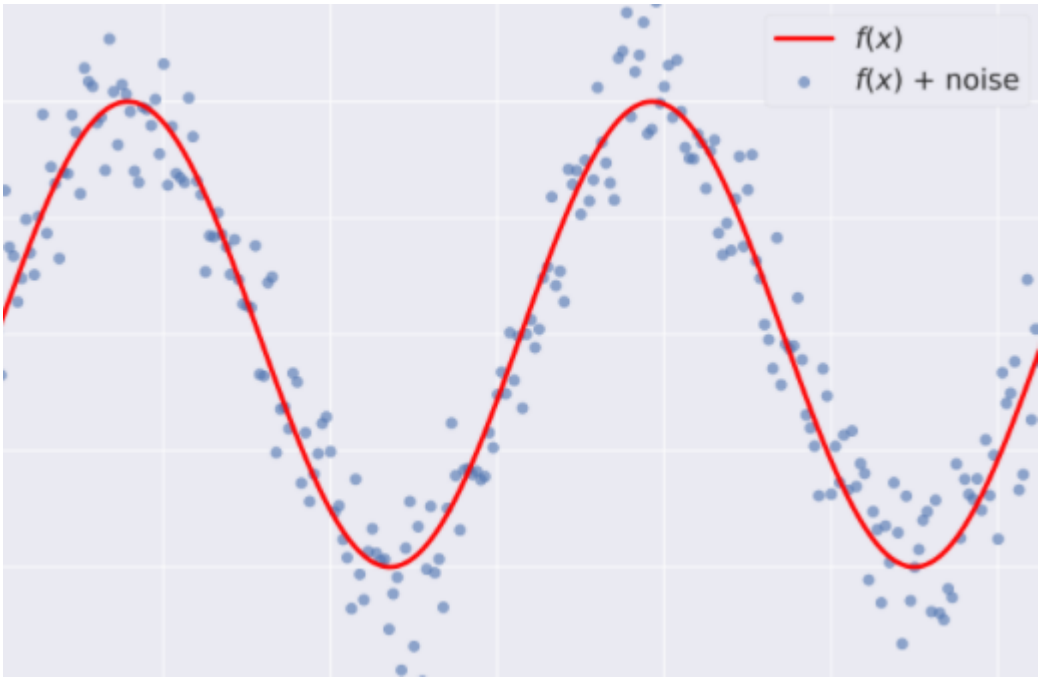
. . .

## Chapter 2. The Bias-Variance Tradeoff

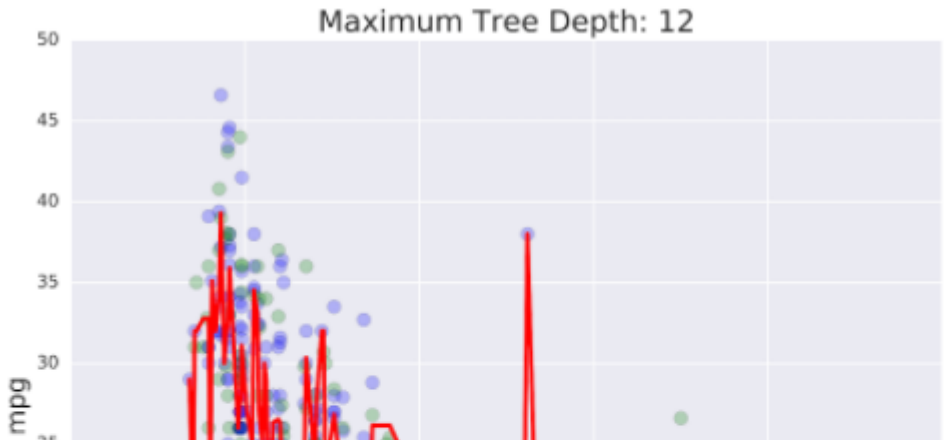
The bias-variance tradeoff is one of the fundamental concepts in supervised machine learning. In this chapter, you'll understand how to diagnose the problems of overfitting and underfitting. You'll also be introduced to the concept of ensembling where the predictions of several models are aggregated to produce predictions that are more robust.

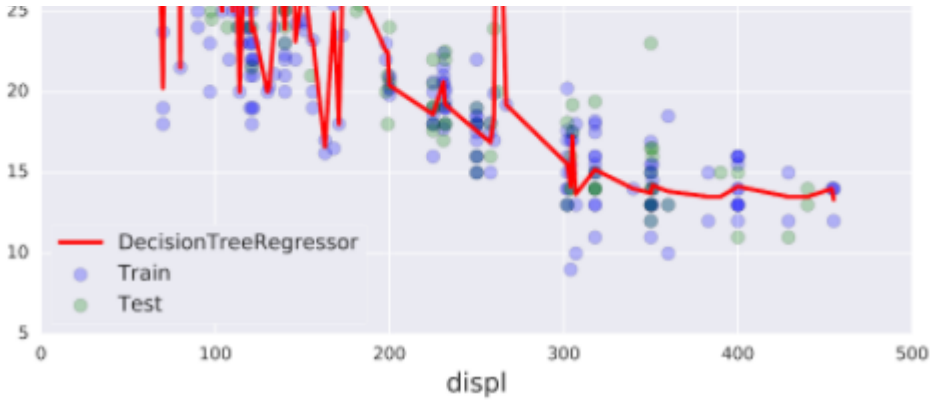
## Generalization Error

In supervised learning, the assumption is made that there is a mapping f between features and labels: y = f(x)

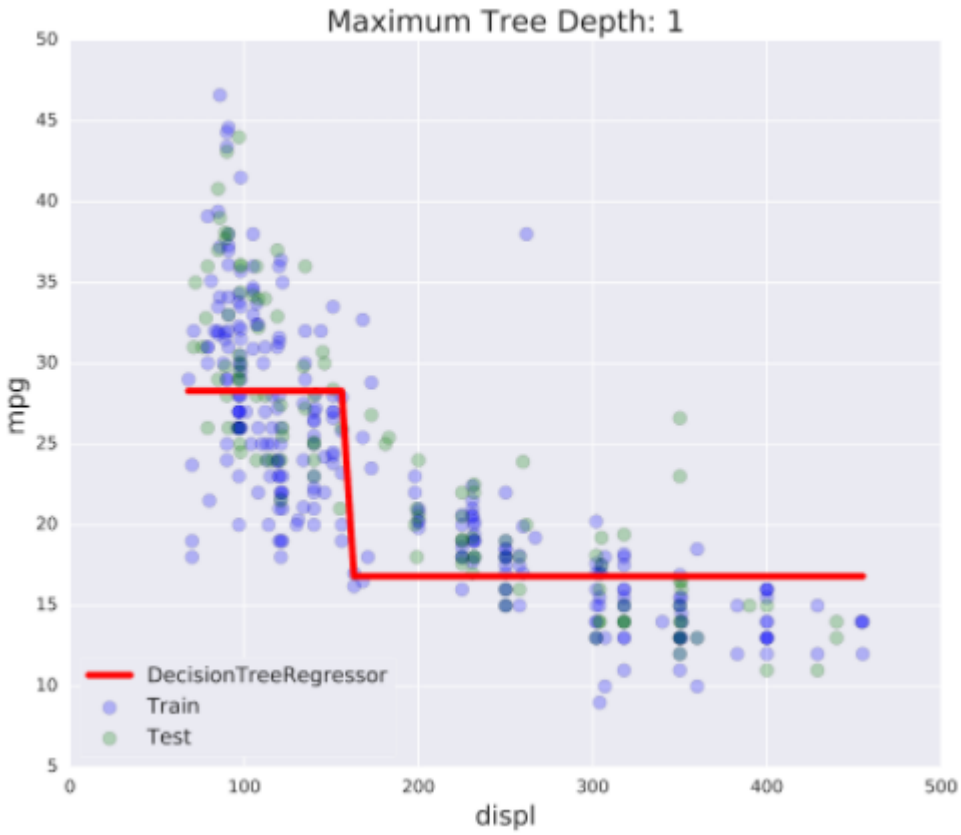



The goal of supervised learning is to **approximate f** (using Logistic Regression, Decision Tree, Neural Network, etc), while discarding noise as much as possible, to achieve a low predictive error on unseen datasets.





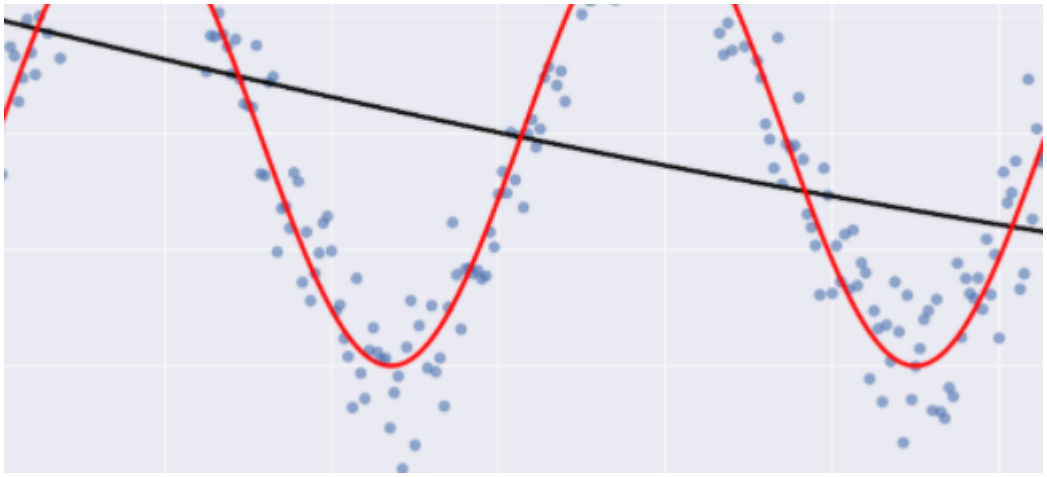

**Overfitting** = low training set error, high test set error




**Underfitting** = high training set error, high test set error

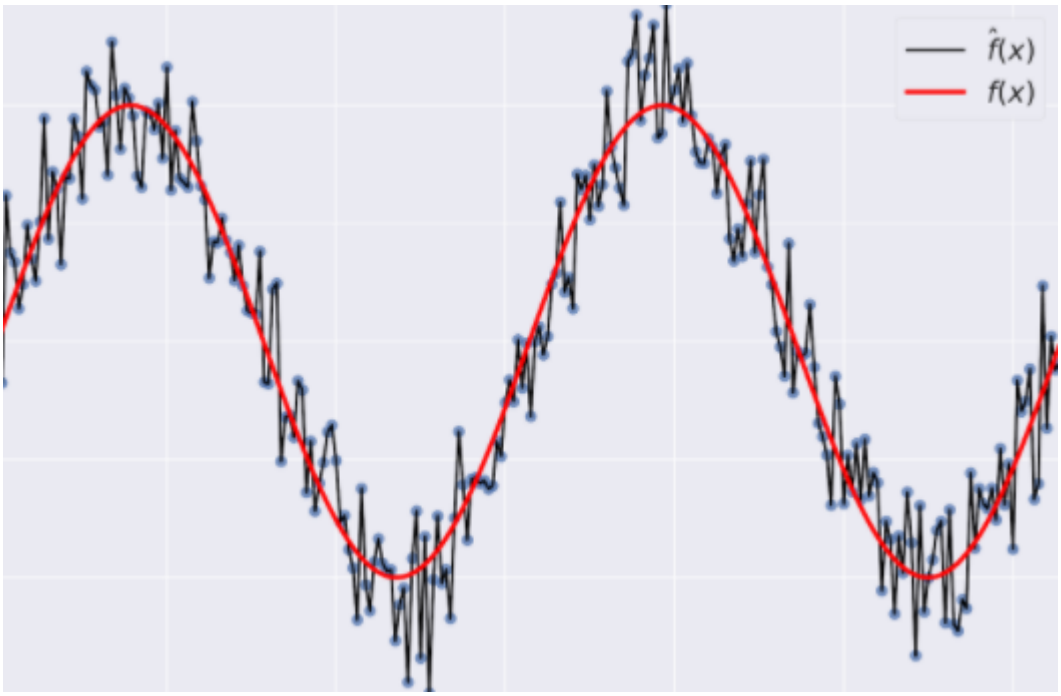
**Generalization Error** (GE) shows how well it generalizes on unseen data.
GE of a model = Bias + Variance + Noise

**Bias** = how different are 'f_approx' and 'f'. High bias models lead to underftting.

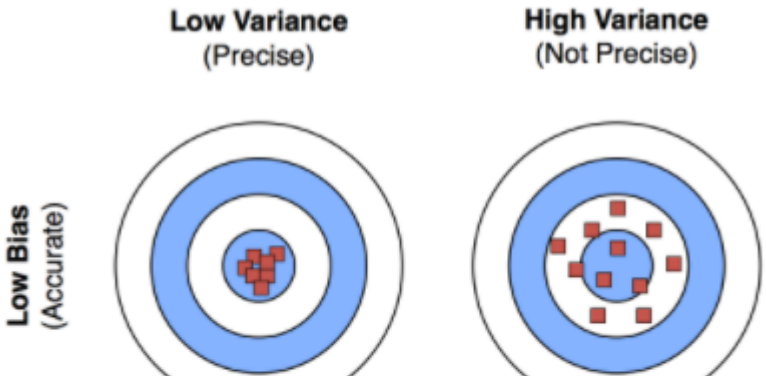


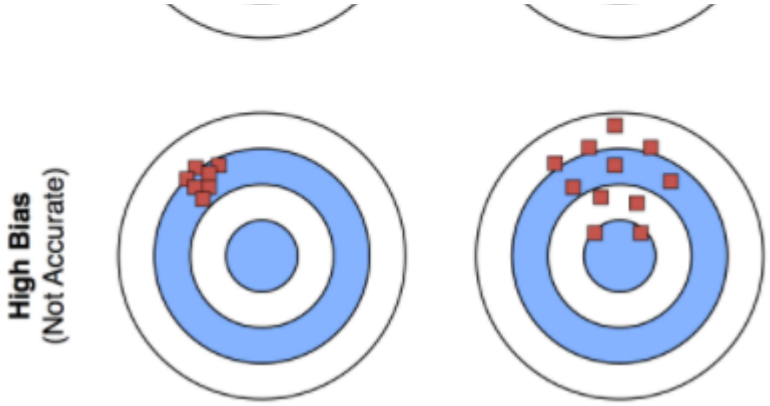
**High bias** model (f_approx) in black, true function (f) in red

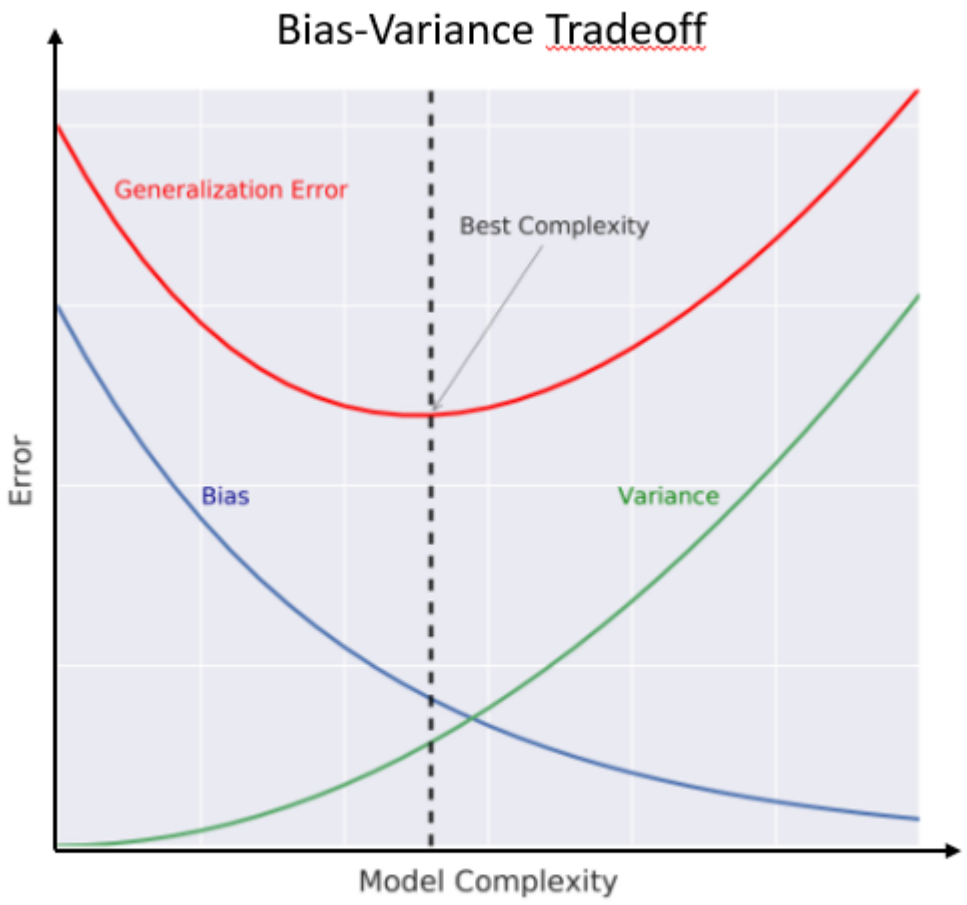
**Variance** = how the 'f_approx' is inconsistent over different training sets. High variance models lead to overfitting.




**High variance** model (f_approx) in black, true function (f) in red






## Complexity, bias and variance

The complexity of a model labeled **f^** influences the bias and variance terms of its generalization error. Which of the following correctly describes the relationship between **f^**'s complexity and **f^**'s bias and variance terms?

- As the complexity of $f$ decreases, the bias term decreases while the variance term increases.

- As the complexity of $\hat{f}$ decreases, both the bias and the variance terms increase.
- As the complexity of $\hat{f}$ increases, the bias term increases while the variance term decreases.
- As the complexity of $\hat{f}$ increases, the bias term decreases while the variance term increases.

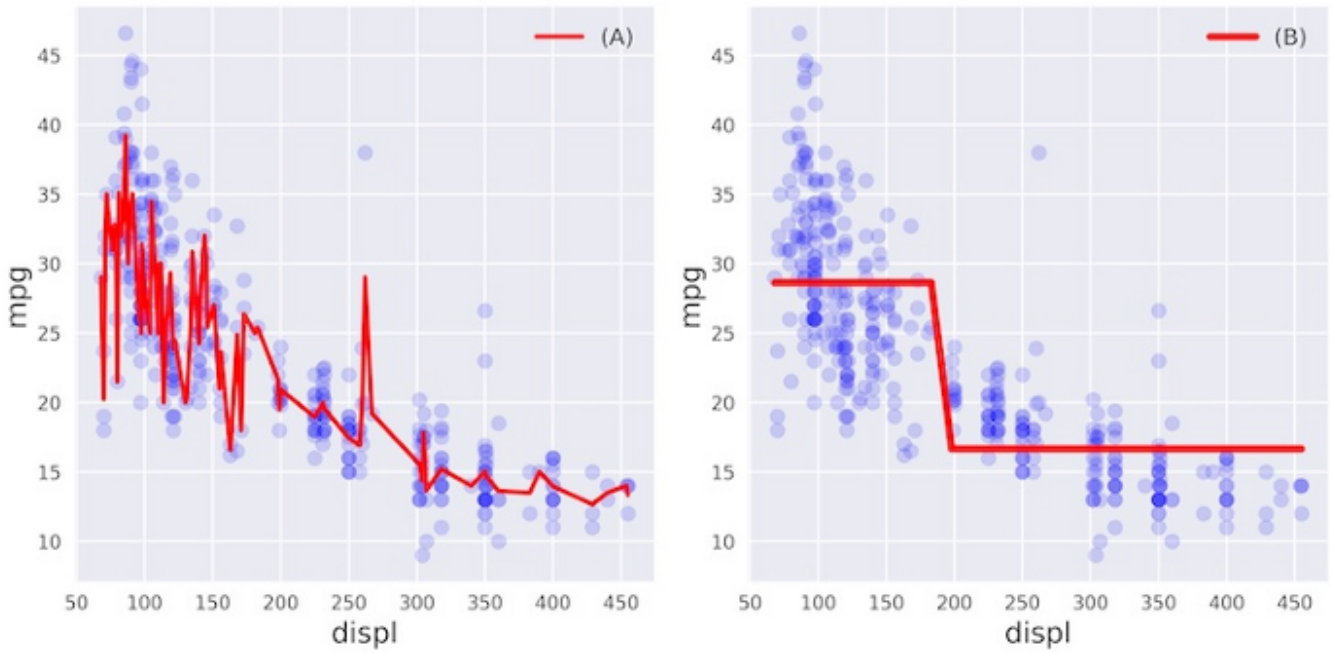
Answer: As the complexity of f^ increases, the bias term decreases while the variance term increases.

## Overfitting and underfitting

In this exercise, you'll visually diagnose whether a model is overfitting or underfitting the training set.

For this purpose, we have trained two different models AA and BB on the auto dataset to predict the `mpg` consumption of a car using only the car's displacement ( `displ` ) as a feature.

The following figure shows you scatterplots of `mpg` versus `displ` along with lines corresponding to the training set predictions of models AA and BB in red.




Which of the following statements is true?

- $A$ suffers from high bias and overfits the training set.

- $A$ suffers from high variance and underfits the training set.
- $B$ suffers from high bias and underfits the training set.
- $B$ suffers from high variance and underfits the training set.

Answer: B suffers from high bias and underfits the training set.
Model B is not able to capture the nonlinear dependence of `mpg` on `displ`.

## Diagnose bias and variance problems

Estimating the Generalization Error (GE) of f_approx by:
1. split the data into training and test sets
2. fit f_approx to the training set
3. evaluate the error of f_approx on the unseen test set
4. GE of f_approx is ≈ test set error of f_approx
5. better with Cross Validation (CV) on training set, eg. k=10, and calculate the average error


**High variance**: high CV-error, low training set error,
ie, **overfit** the training set.
Remedy: decrease model complexity (decrease max_depth, increase min_samples_leaf, etc), reduce number of features or gather more data.

High bias: CV-error ≈ training set error, and both are high

ie, underfit the training set.

Remedy: increase model complexity (increase max_depth, decrease min_samples_leaf, etc), gather more relevant features.

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error as MSE
from sklearn.model_selection import cross_val_score

# Set seed for reproducibility
SEED = 123
# Split data into 70% train and 30% test
X_train, X_test, y_train, y_test = train_test_split(X,y,
                                                    test_size=0.3,
                                                    random_state=SEED)

# Instantiate decision tree regressor and assign it to 'dt'
dt = DecisionTreeRegressor(max_depth=4,
                           min_samples_leaf=0.14,
                           random_state=SEED)

# Evaluate the list of MSE ontained by 10-fold CV
# Set n_jobs to -1 in order to exploit all CPU cores in computation
MSE_CV = - cross_val_score(dt, X_train, y_train, cv= 10,
                           scoring='neg_mean_squared_error',
                           n_jobs = -1)

# Fit 'dt' to the training set
dt.fit(X_train, y_train)
# Predict the labels of training set
y_predict_train = dt.predict(X_train)
# Predict the labels of test set
y_predict_test = dt.predict(X_test)

# CV MSE
print('CV MSE: {:.2f}'.format(MSE_CV.mean()))
```

```
CV MSE: 20.51
```

```
# Training set MSE
print('Train MSE: {:.2f}'.format(MSE(y_train, y_predict_train)))
```

```
Train MSE: 15.30
```

```
# Test set MSE
print('Test MSE: {:.2f}'.format(MSE(y_test, y_predict_test)))
```

```
Test MSE: 20.92
```

Given that the CV-error (20.51) is higher than the training set error (15.30), we can deduce that dt overfits the training set and suffers from high variance.

## Instantiate the model

In the following set of exercises, you'll diagnose the bias and variance problems of a regression tree. The regression tree you'll define in this exercise will be used to predict the mpg consumption of cars from the auto dataset using all available features.

We have already processed the data and loaded the features matrix `X` and the array `y` in your workspace. In addition, the `DecisionTreeRegressor` class was imported from `sklearn.tree`.

```
# Import train_test_split from sklearn.model_selection
from sklearn.model_selection import train_test_split

# Set SEED for reproducibility
SEED = 1

# Split the data into 70% train and 30% test
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=SEED)

# Instantiate a DecisionTreeRegressor dt
dt = DecisionTreeRegressor(max_depth=4, min_samples_leaf=0.26,
random_state=SEED)
```

## Evaluate the 10-fold CV error

In this exercise, you'll evaluate the 10-fold CV Root Mean Squared Error (RMSE) achieved by the regression tree `dt` that you instantiated in the previous exercise.

In addition to `dt`, the training data including `X_train` and `y_train` are available in your workspace. We also imported `cross_val_score` from `sklearn.model_selection`.

Note that since `cross_val_score` has only the option of evaluating the negative MSEs, its output should be multiplied by negative one to obtain the MSEs. The CV RMSE can then be obtained by computing the square root of the average MSE.

```
# Compute the array containing the 10-folds CV MSEs
MSE_CV_scores = - cross_val_score(dt, X_train, y_train, cv=10,
                       scoring='neg_mean_squared_error',
                       n_jobs=-1)

# Compute the 10-folds CV RMSE
RMSE_CV = (MSE_CV_scores.mean())**(0.5)

# Print RMSE_CV
print('CV RMSE: {:.2f}'.format(RMSE_CV))
```

```
<script.py> output:
    CV RMSE: 5.14
```

A very good practice is to keep the test set untouched until you are confident about your model's performance. CV is a great technique to get an estimate of a model's performance without affecting the test set.

### Evaluate the training error

You'll now evaluate the training set RMSE achieved by the regression tree `dt` that you instantiated in a previous exercise.

In addition to `dt`, `X_train` and `y_train` are available in your workspace.

Note that in scikit-learn, the MSE of a model can be computed as follows:

```
MSE_model = mean_squared_error(y_true, y_predicted)
```

where we use the function `mean_squared_error` from the `metrics` module and pass it the true labels `y_true` as a first argument, and the predicted labels from the model `y_predicted` as a second argument.

```
# Import mean_squared_error from sklearn.metrics as MSE
from sklearn.metrics import mean_squared_error as MSE

# Fit dt to the training set
dt.fit(X_train, y_train)
```

```
# Predict the labels of the training set
y_pred_train = dt.predict(X_train)

# Evaluate the training set RMSE of dt
RMSE_train = (MSE(y_train, y_pred_train))**(0.5)

# Print RMSE_train
print('Train RMSE: {:.2f}'.format(RMSE_train))
```

```
<script.py> output:
    Train RMSE: 5.15
```

Notice how the training error is roughly equal to the 10-folds CV error you obtained in the previous exercise.

### High bias or high variance?

In this exercise you'll diagnose whether the regression tree `dt` you trained in the previous exercise suffers from a bias or a variance problem.

Achieved by by `dt`, the training set RMSE is `RMSE_train` and the CV-RMSE is `RMSE_CV`. An additional variable called `baseline_RMSE` is the root mean-squared error achieved by the regression-tree trained with the `disp` feature only (it is the RMSE achieved by the regression tree trained in chapter 1, lesson 3). Here `baseline_RMSE` serves as the baseline RMSE above which a model is considered to be underfitting and below which the model is considered 'good enough'.

```
In [1]: baseline_RMSE
Out[1]: 5.1

In [2]: RMSE_CV
Out[2]: 5.14

In [3]: RMSE_train
Out[3]: 5.15
```

Does `dt` suffer from a high bias or a high variance problem?

- `dt` suffers from high variance because `RMSE_CV` is far less than `RMSE_train`.

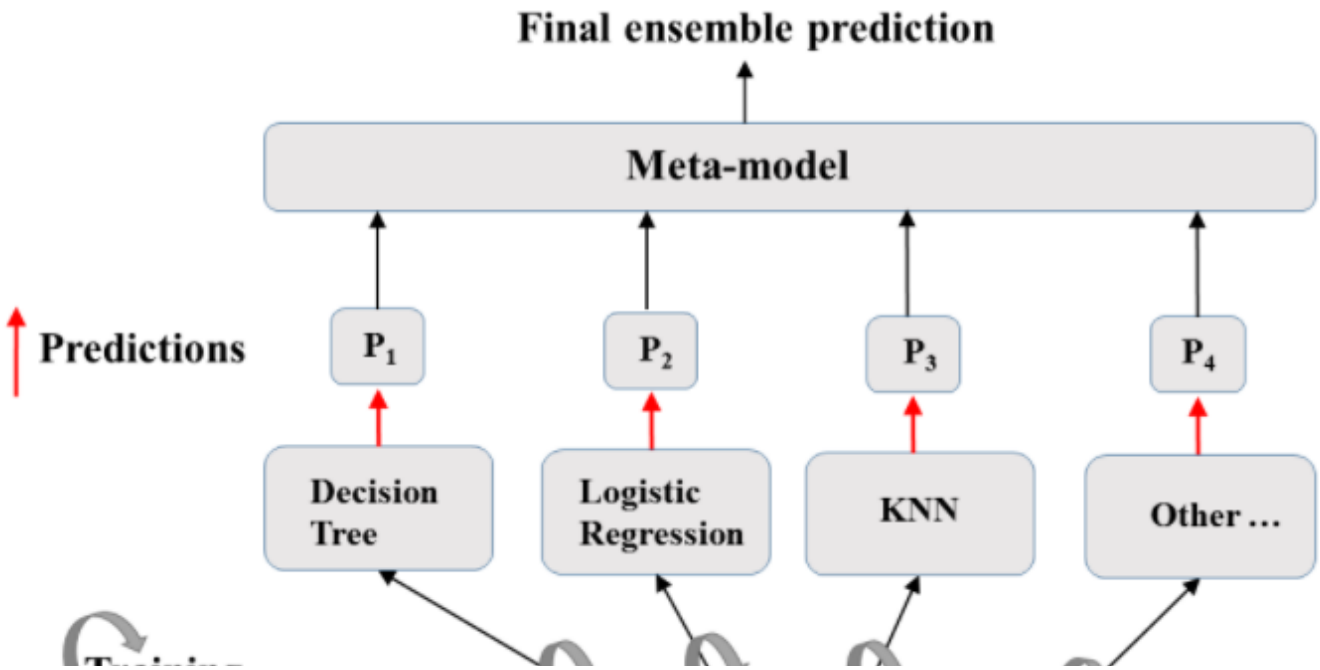
- `dt` suffers from high bias because `RMSE_CV` ≈ `RMSE_train` and both scores are greater than `baseline_RMSE` .
- `dt` is a good fit because `RMSE_CV` ≈ `RMSE_train` and both scores are smaller than `baseline_RMSE` .

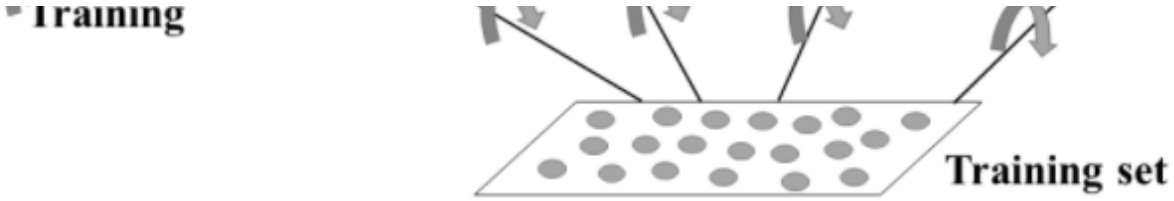
Answer: `dt` suffers from high bias because `RMSE_CV` ≈ `RMSE_train` and both scores are greater than `baseline_RMSE` . The model is underfitting the training set as the model is too constrained to capture the nonlinear dependencies between features and labels.

## Ensemble Learning

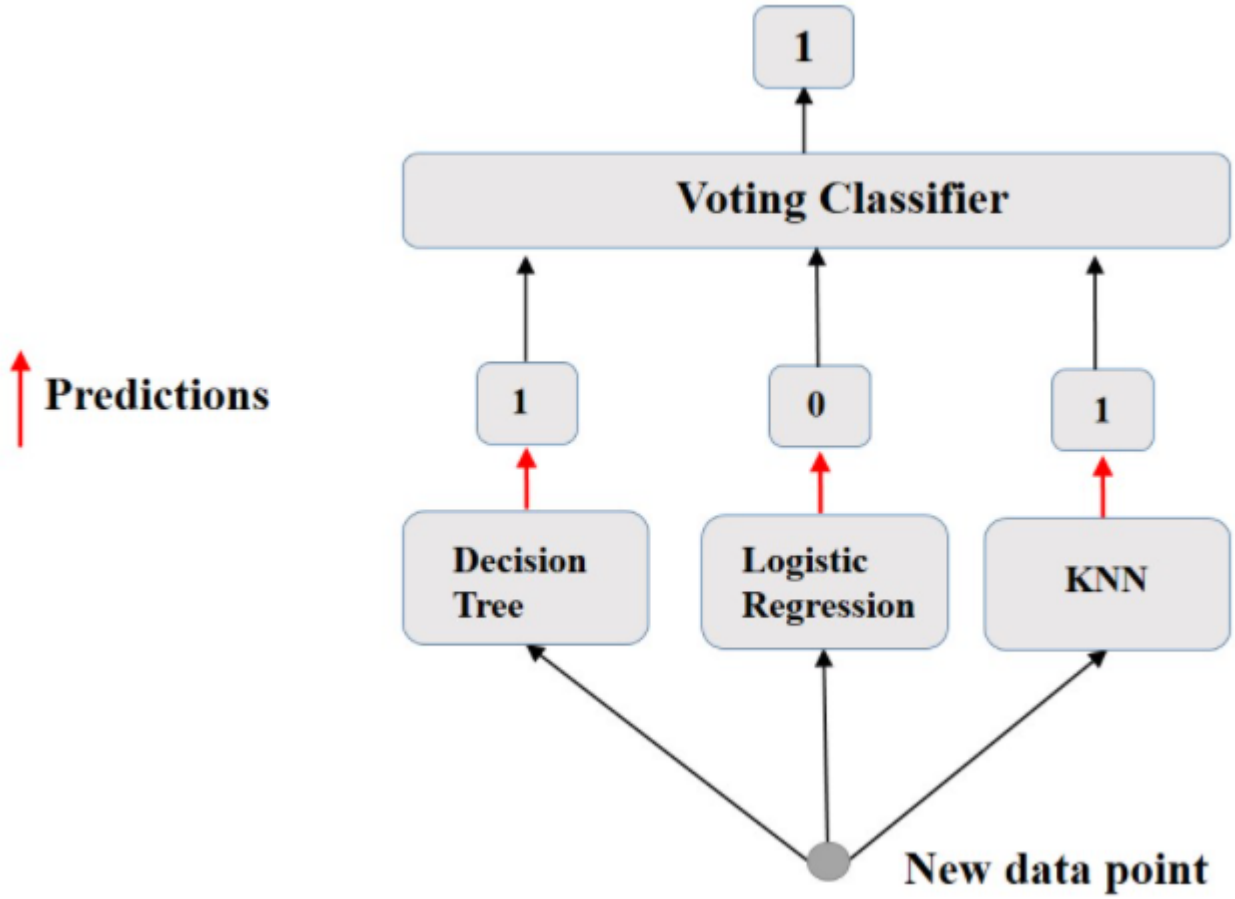
Limitations of Classification & Regression Trees (CARTs)
· can only produce orthogonal decision boundaries
· sensitive to small variations in the training set, tendency to memorise noise
· **high variance**: unconstrained CARTS may **overfit** the training set
Solution: ensemble learning

Steps for Ensemble Learning:
1. Train different models on the same dataset
2. Let each model make its predictions
3. Meta-model: aggregates predictions of individual models and outputs the final prediction.
The final prediction is more robust and less prone to errors. The best results are obtained as models are skillful in different ways. If some models make predictions that are way off, the other models should compensate these errors.






Ensemble Learning




Voting Classifier

```
# Import functions to compute accuracy and split data
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

# Import models, including VotingClassifier meta-model
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier as KNN
from sklearn.ensemble import VotingClassifier

# Set seed for reproducibility
SEED = 1
# Split data into 70% train and 30% test
X_train, X_test, y_train, y_test = train_test_split(X, y
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size= 0.3,
                                                    random_state= SEED)
# Instantiate individual classifiers
lr = LogisticRegression(random_state=SEED)
knn = KNN()
dt = DecisionTreeClassifier(random_state=SEED)
# Define a list called classifier that contains the tuples (classifier_name, classifier)
classifiers = [('Logistic Regression', lr),
               ('K Nearest Neighbours', knn),
               ('Classification Tree', dt)]
# Iterate over the defined list of tuples containing the classifiers
for clf_name, clf in classifiers:
    #fit clf to the training set
    clf.fit(X_train, y_train)

    # Predict the labels of the test set
    y_pred = clf.predict(X_test)

    # Evaluate the accuracy of clf on the test set
    print('{:s} : {:.3f}'.format(clf_name, accuracy_score(y_test, y_pred)))
```

```
Logistic Regression: 0.947
K Nearest Neighbours: 0.930
Classification Tree: 0.930
```

Results from individual classifiers

```
# Instantiate a VotingClassifier 'vc'
vc = VotingClassifier(estimators=classifiers)

# Fit 'vc' to the traing set and predict test set labels
vc.fit(X_train, y_train)
y_pred = vc.predict(X_test)

# Evaluate the test-set accuracy of 'vc'
print('Voting Classifier: {.3f}'.format(accuracy_score(y_test, y_pred)))
```

```
Voting Classifier: 0.953
```

Results from VotingClassifier 'vc' is higher than any individual model

## Define the ensemble

In the following set of exercises, you'll work with the **Indian Liver Patient Dataset** from the UCI Machine learning repository.

In this exercise, you'll instantiate three classifiers to predict whether a patient suffers from a liver disease using all the features present in the dataset.

```
# Set seed for reproducibility
SEED=1

# Instantiate lr
lr = LogisticRegression(random_state=SEED)

# Instantiate knn
knn = KNeighborsClassifier(n_neighbors=27)

# Instantiate dt
dt = DecisionTreeClassifier(min_samples_leaf=0.13, random_state=SEED)

# Define the list classifiers
classifiers = [('Logistic Regression', lr), ('K Nearest Neighbours',
knn), ('Classification Tree', dt)]
```

## Evaluate individual classifiers

In this exercise you'll evaluate the performance of the models in the list `classifiers` that we defined in the previous exercise. You'll do so by fitting each classifier on the training set and evaluating its test set accuracy.

The dataset is already loaded and preprocessed for you (numerical features are standardized) and it is split into 70% train and 30% test.

```
# Iterate over the pre-defined list of classifiers
for clf_name, clf in classifiers:

    # Fit clf to the training set
    clf.fit(X_train, y_train)

    # Predict y_pred
    y_pred = clf.predict(X_test)

    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)
```

```
# Evaluate clf's accuracy on the test set
print('{:s} : {:.3f}'.format(clf_name, accuracy))
```

```
<script.py> output:
    Logistic Regression : 0.747
    K Nearest Neighbours : 0.724
    Classification Tree : 0.730
```

Logistic Regression achieved the highest accuracy of 74.7%

## Better performance with a Voting Classifier

Finally, you'll evaluate the performance of a voting classifier that takes the outputs of the models defined in the list `classifiers` and assigns labels by majority voting.

```
# Import VotingClassifier from sklearn.ensemble
from sklearn.ensemble import VotingClassifier

# Instantiate a VotingClassifier vc
vc = VotingClassifier(estimators=classifiers)

# Fit vc to the training set
vc.fit(X_train, y_train)

# Evaluate the test set predictions
y_pred = vc.predict(X_test)

# Calculate accuracy score
accuracy = accuracy_score(y_test, y_pred)
print('Voting Classifier: {:.3f}'.format(accuracy))
```

```
<script.py> output:
    Voting Classifier: 0.753
```

Note: The voting classifier achieves a test set accuracy of 75.3%. This value is greater than that previously achieved by `LogisticRegression` (74.7%).

. . .

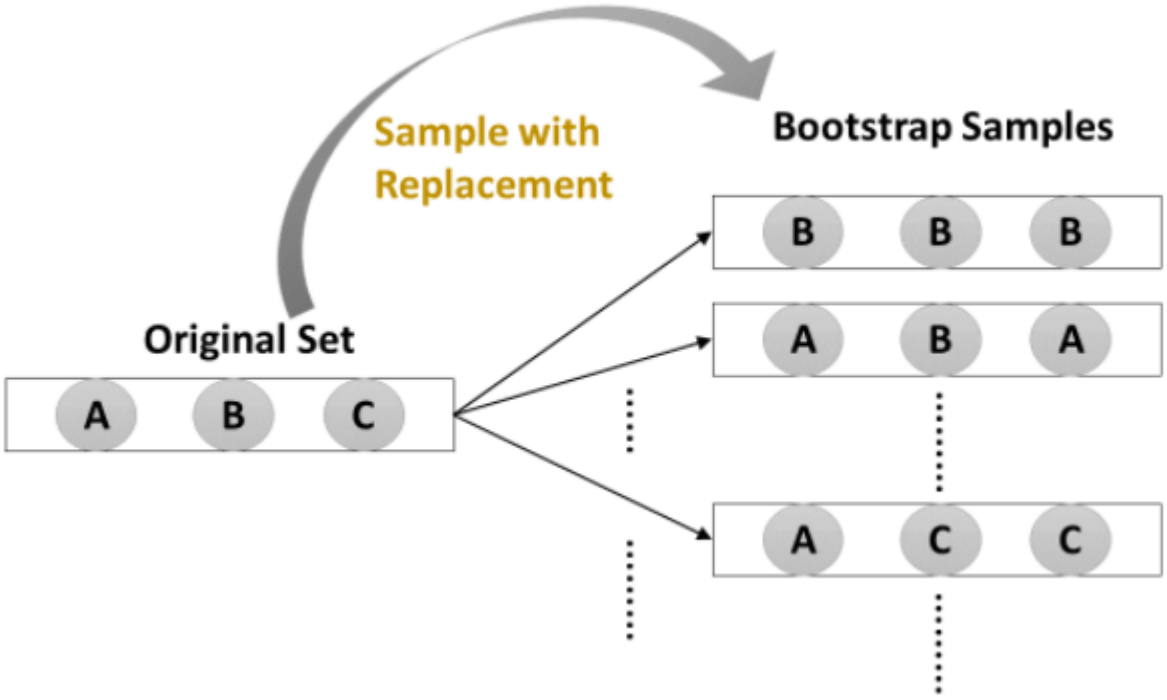
# Chapter 3. Bagging and Random Forests

Bagging is an ensemble method involving training the same algorithm many times using different subsets sampled from the training data. In this chapter, you'll understand how bagging can be used to create a tree ensemble. You'll also learn how the random forests algorithm can lead to further ensemble diversity through randomization at the level of each split in the trees forming the ensemble.

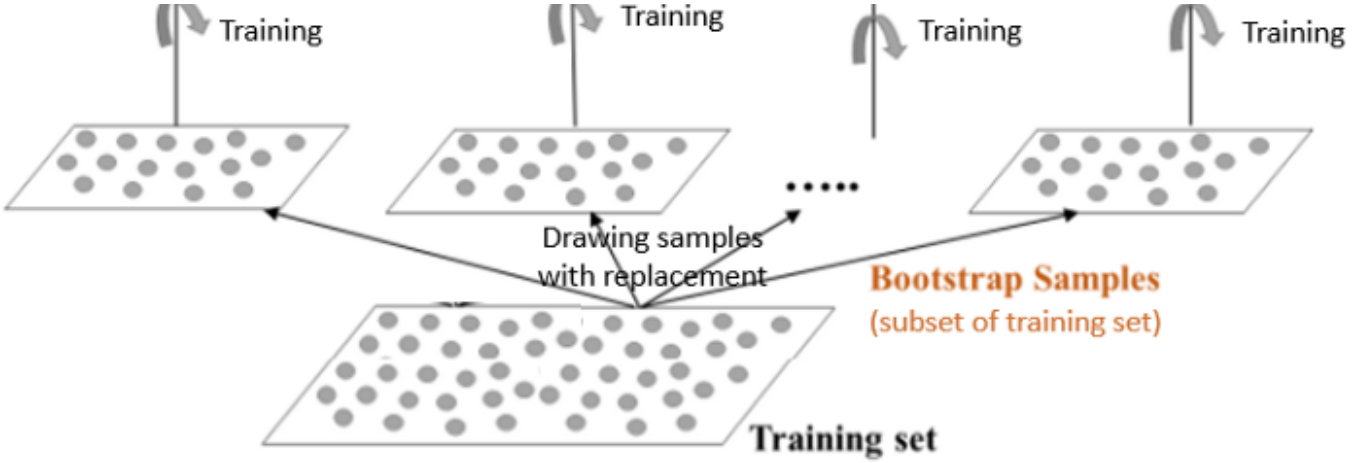
## Bagging

Bagging stands for Bootstrap Aggregation, and uses a technique known as the bootstrap. Bagging has the effect of reducing the variance of individual models in the ensemble.

| | Voting Classifier | Bagging |
|---|---|---|
| Algorithm | Many different algorithms | One algorithm |
| Training data | Same (whole) training set | Many subsets of the training set |




Bagging consists of drawing N different bootstrap samples from the training set.




Drawing N bootstrap samples to train N models (that use the same algorithm)

Each model outputs its prediction to be compiled in a bagging ensemble. The Meta Model collects predictions of all N models, and outputs a final prediction.
In classification, the final prediction is obtained by majority voting, using the `BaggingClassifier` in scikit-learn.
In regression, the final prediction is the average of the N model predictions, using the `BaggingRegressor` in scikit-learn.

```
# Import models and utility functions
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

# Set seed for reproducibility
SEED = 1

# Split data into 70% train and 30% test
X_train, X_test, y_train, y_test = train_test_split(X, y,
                test_size=0.3, stratify=y, random_state=SEED)

# Instantiate a classification-tree 'dt'
dt = DecisionTreeClassifier(max_depth=4,
                min_samples_leaf=0.16, random_state=SEED)

# Instantiate a BaggingClassifier 'bc'
bc = BaggingClassifier(base_estimator=dt,
                n_estimators=300, n_jobs=-1)

# Fit 'bc' to the training set
bc.fit(X_train, y_train)
```

```
# Predict test set labels
y_pred = bc.predict(X_test)

# Evaluate and print test-set accuracy
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy of Bagging Classifier: {:.3f}'.format(accuracy))
```

```
Accuracy of Bagging Classifier: 0.936
```

In the beginning of Chapter 1, dt achieved an accuracy of 90.3%. The accuracy achieved by the same model dt using bagging ensemble is 93.6%. Bagging outperforms the base estimator dt.

## Define the bagging classifier

In the following exercises you'll work with the **Indian Liver Patient** dataset from the UCI machine learning repository. Your task is to predict whether a patient suffers from a liver disease using 10 features including Albumin, age and gender. You'll do so using a Bagging Classifier.

```
# Import DecisionTreeClassifier
from sklearn.tree import DecisionTreeClassifier

# Import BaggingClassifier
from sklearn.ensemble import BaggingClassifier

# Instantiate dt
dt = DecisionTreeClassifier(random_state=1)

# Instantiate bc
bc = BaggingClassifier(base_estimator=dt, n_estimators=50,
random_state=1)
```

## Evaluate Bagging performance

Now that you instantiated the bagging classifier, it's time to train it and evaluate its test set accuracy.

The Indian Liver Patient dataset is processed for you and split into 80% train and 20% test. The feature matrices `X_train` and `X_test`, as well as the arrays of labels `y_train`

and `y_test` are available in your workspace. In addition, we have also loaded the bagging classifier `bc` that you instantiated in the previous exercise and the function `accuracy_score()` from `sklearn.metrics`.

```
# Fit bc to the training set
bc.fit(X_train, y_train)

# Predict test set labels
y_pred = bc.predict(X_test)

# Evaluate acc_test
acc_test = accuracy_score(y_test, y_pred)
print('Test set accuracy of bc: {:.2f}'.format(acc_test))
```

```
<script.py> output:
    Test set accuracy of bc: 0.71
```

```
# Cross check dt performance
dt.fit(X_train, y_train)
y_pred = dt.predict(X_test)
acc_test = accuracy_score(y_test, y_pred)
print('Test set accuracy of dt: {:.2f}'.format(acc_test))
```

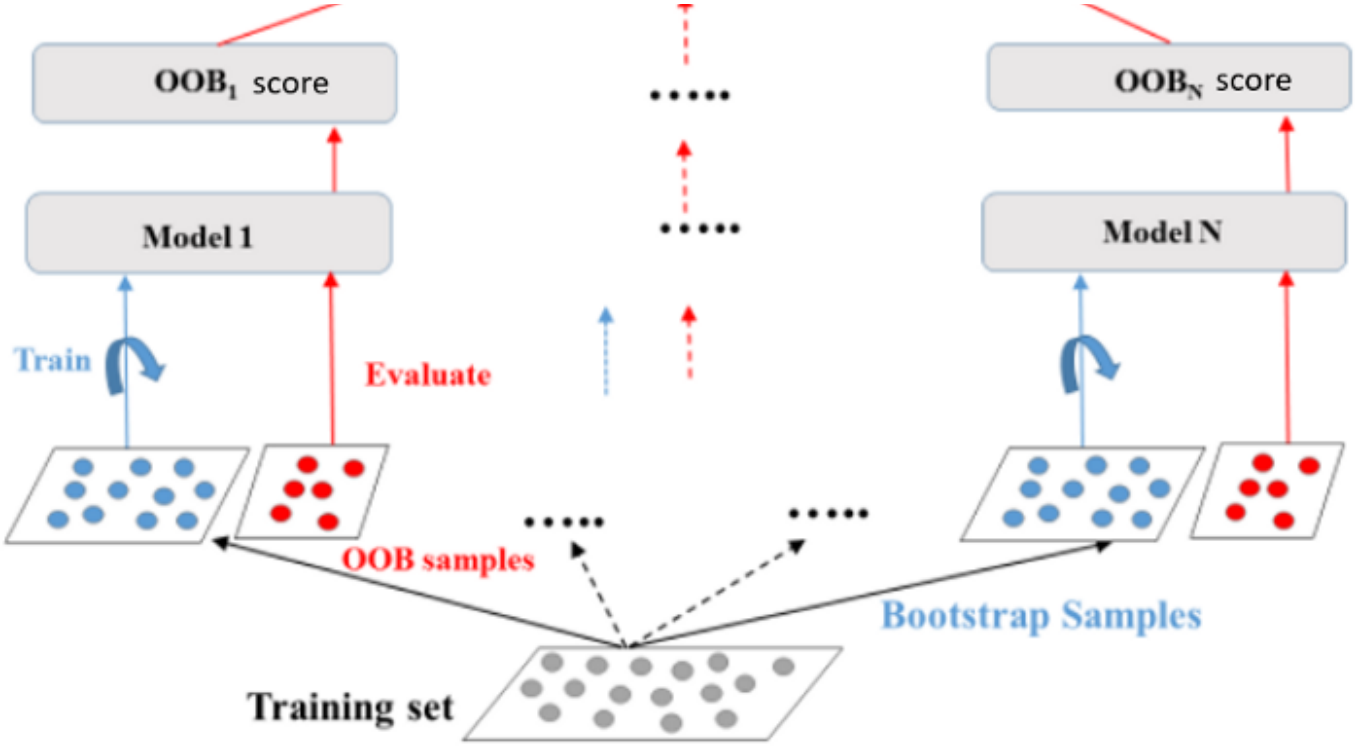
```
<script.py> output:
    Test set accuracy of dt: 0.63
```

A single tree `dt` would have achieved an accuracy of 63% which is 8% lower than `bc`'s accuracy!

## Out of Bag (OOB) Evaluation

OOB instances are training instances that are not sampled and not seen by the model during training. These can be used to estimate the performance of the ensemble without the need for cross-validation.

$$OOB\ \text{score} = \frac{OOB_1 + \ldots + OOB_N}{N}$$




OOB evaluation

```
# Import models and split utility function
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

# Set seed for reproducibility
SEED = 1

# Split data into 70% train and 30% test
X_train, X_test, y_train, y_test = train_test_split(X, y,
            test_size= 0.3, stratify= y, random_state=SEED)

# Instantiate a classification-tree 'dt'
dt = DecisionTreeClassifier(max_depth=4,
            min_samples_leaf=0.16, random_state=SEED)

# Instantiate a BaggingClassifier 'bc'; set oob_score= True
bc = BaggingClassifier(base_estimator=dt,
            n_estimators=300, oob_score=True, n_jobs=-1)

# Fit 'bc' to the traing set
bc.fit(X_train, y_train)

# Predict the test set labels
y_pred = bc.predict(X_test)
```

```
# Evaluate test set accuracy
test_accuracy = accuracy_score(y_test, y_pred)

# Extract the OOB accuracy from 'bc'
oob_accuracy = bc.oob_score_

# Print test set accuracy
print('Test set accuracy: {:.3f}'.format(test_accuracy))

# Print OOB accuracy
print('OOB accuracy: {:.3f}'.format(oob_accuracy)
```

```
# Print test set accuracy
print('Test set accuracy: {:.3f}'.format(test_accuracy))
```

```
Test set accuracy: 0.936
```

```
# Print OOB accuracy
print('OOB accuracy: {:.3f}'.format(oob_accuracy))
```

```
OOB accuracy: 0.925
```

The 2 obtained accuracies are close, so OOB evaluation an be an efficient technique to obtain a performance estimate of a bagged ensemble on unseen data without cross validation.

## Prepare the ground

In the following exercises, you'll compare the OOB accuracy to the test set accuracy of a bagging classifier trained on the Indian Liver Patient dataset.

In sklearn, you can evaluate the OOB accuracy of an ensemble classifier by setting the parameter `oob_score` to `True` during instantiation. After training the classifier, the OOB accuracy can be obtained by accessing the `.oob_score_` attribute from the corresponding instance.

```
# Import DecisionTreeClassifier
from sklearn.tree import DecisionTreeClassifier

# Import BaggingClassifier
from sklearn.ensemble import BaggingClassifier
```

```
# Instantiate dt
dt = DecisionTreeClassifier(min_samples_leaf=8, random_state=1)

# Instantiate bc
bc = BaggingClassifier(base_estimator=dt,
            n_estimators=50,
            oob_score=True,
            random_state=1)
```

### OOB Score vs Test Set Score

Now that you instantiated `bc`, you will fit it to the training set and evaluate its test set and OOB accuracies.

The dataset is processed for you and split into 80% train and 20% test. The feature matrices `X_train` and `X_test`, as well as the arrays of labels `y_train` and `y_test` are available in your workspace. In addition, we have also loaded the classifier `bc` instantiated in the previous exercise and the function `accuracy_score()` from `sklearn.metrics`.

```
# Fit bc to the training set
bc.fit(X_train, y_train)

# Predict test set labels
y_pred = bc.predict(X_test)

# Evaluate test set accuracy
acc_test = accuracy_score(y_test, y_pred)

# Evaluate OOB accuracy
acc_oob = bc.oob_score_

# Print acc_test and acc_oob
print('Test set accuracy: {:.3f}, OOB accuracy:
{:.3f}'.format(acc_test, acc_oob))
```

```
<script.py> output:
    Test set accuracy: 0.698, OOB accuracy: 0.704
```

The test set accuracy and the OOB accuracy of `bc` are both roughly equal to 70%!

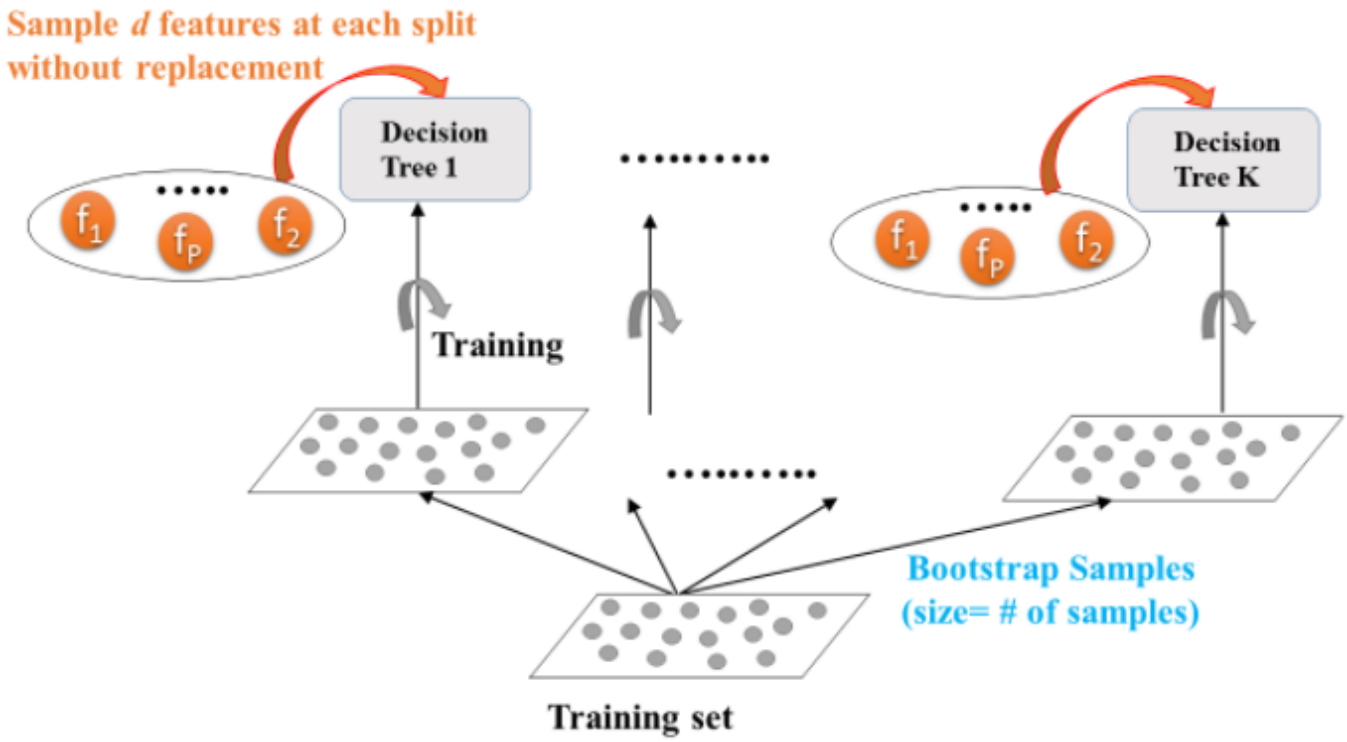
## Random Forests (RF)

RF is another ensemble learning method.

Base estimator: Decision Tree

Each estimator is trained on a different bootstrap sample having the same size as the training set. RF introduces further randomization than bagging when training each of the base estimators.

When each tree is trained, only d features can be sampled at each node without replacement (d < total number of features).




Random Forests training

Each base estimator outputs its prediction to be compiled in a bagging ensemble. The **Meta Model** collects predictions of all N models, and outputs a final prediction.

In classification, the final prediction is obtained by majority voting, using the `RandomForestClassifier` in scikit-learn.

In regression, the final prediction is the average of the N model predictions, using the `RandomForestRegressor` in scikit-learn.

In general, Random Forests achieves a lower variance than individual trees.

```
# Basic imports
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error as MSE

# Set seed for reproducibility
SEED = 1

# Split dataset into 70% train and 30% test
X_train, X_test, y_train, y_test = train_test_split(X, y,
            test_size=0.3, random_state=SEED)

# Instantiate a random forests regressor 'rf' 400 estimators
rf = RandomForestRegressor(n_estimators=400,
            min_samples_leaf=0.12, random_state=SEED)

# Fit 'rf' to the training set
rf.fit(X_train, y_train)

# Predict the test set labels 'y_pred'
y_pred = rf.predict(X_test)

# Evaluate the test set RMSE
rmse_test = MSE(y_test, y_pred)**(1/2)

# Print the test set RMSE
print('Test set RMSE of rf: {:.2f}'.format(rmse_test))
```

```
# Print the test set RMSE
print('Test set RMSE of rf: {:.2f}'.format(rmse_test))
```

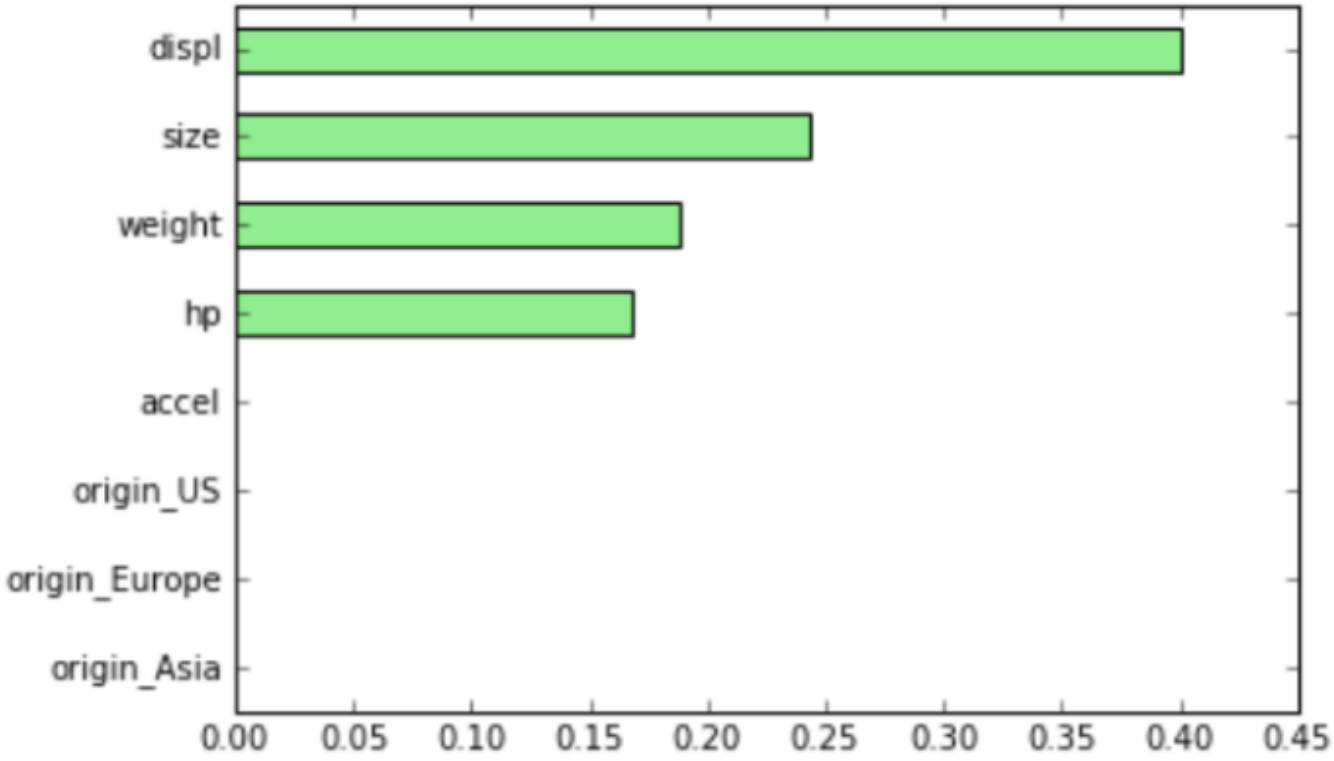
```
Test set RMSE of rf: 3.98
```

rf achieves a test set RMSE of 3.98. This error is smaller than that achieved by a single regression tree (4.43).

When a tree based method is trained, the predictive power of a feature (feature importance) can be assessed, my measuring how much the tree nodes use a particular feature to reduce impurity.
Feature importance is calculated as a percentage indicating the weight of that feature in training and prediction.

```
# FEATURE IMPORTANCE in sklearn
import pandas as pd
import matplotlib.pyplot as plt
```

```
# Create a pd.Series of features importances
importances_rf = pd.Series(rf.feature_importances_,
            index = X.columns)

# Sort importances_rf
sorted_importances_rf = importances_rf.sort_values()

# Make a horizontal bar plot
sorted_importances_rf.plot(kind= 'barh', color= 'lightgreen');
plt.show()
```




## Train an RF regressor

In the following exercises you'll predict bike rental demand in the Capital Bikeshare program in Washington, D.C using historical weather data from the **Bike Sharing Demand** dataset available through Kaggle. For this purpose, you will be using the random forests algorithm. As a first step, you'll define a random forests regressor and fit it to the training set.

The dataset is processed for you and split into 80% train and 20% test.

```
# Import RandomForestRegressor
from sklearn.ensemble import RandomForestRegressor
```

```
# Instantiate rf
rf = RandomForestRegressor(n_estimators=25,
            random_state=2)

# Fit rf to the training set
rf.fit(X_train, y_train)
```

### Evaluate the RF regressor

You'll now evaluate the test set RMSE of the random forests regressor `rf` that you trained in the previous exercise.

The dataset is processed for you and split into 80% train and 20% test. The features matrix `X_test`, as well as the array `y_test` are available in your workspace. In addition, we have also loaded the model `rf` that you trained in the previous exercise.

```
# Import mean_squared_error as MSE
from sklearn.metrics import mean_squared_error as MSE

# Predict the test set labels
y_pred = rf.predict(X_test)

# Evaluate the test set RMSE
rmse_test = MSE(y_test, y_pred)**(1/2)

# Print rmse_test
print('Test set RMSE of rf: {:.2f}'.format(rmse_test))
```

```
<script.py> output:
    Test set RMSE of rf: 51.97
```

You can try training a single CART on the same dataset. The test set RMSE achieved by `rf` is significantly smaller than that achieved by a single CART!

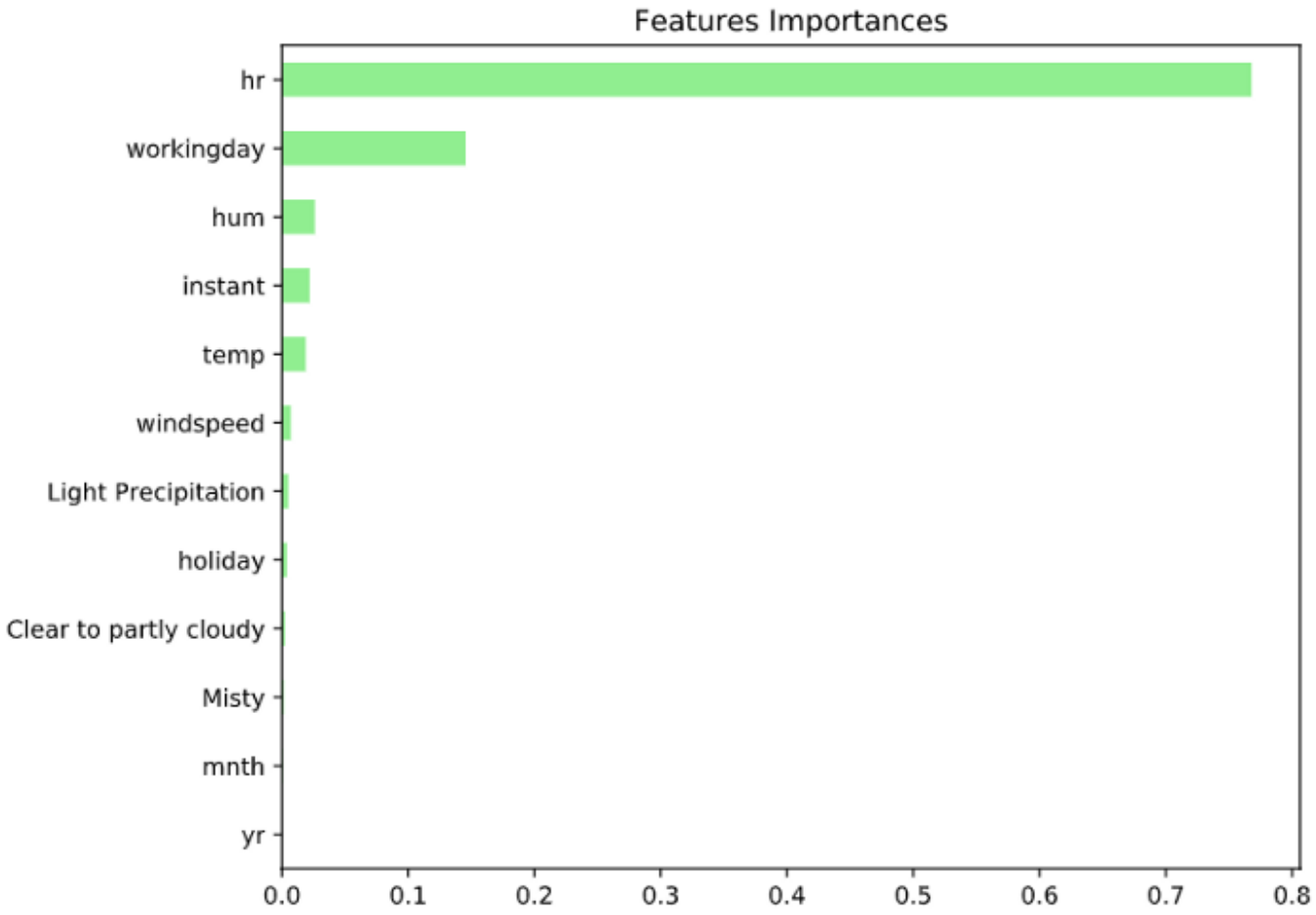
### Visualizing features importances

In this exercise, you'll determine which features were the most predictive according to the random forests regressor `rf` that you trained in a previous exercise.

For this purpose, you'll draw a horizontal barplot of the feature importance as assessed by `rf`. Fortunately, this can be done easily thanks to plotting capabilities of `pandas`.

We have created a `pandas.Series` object called `importances` containing the feature names as `index` and their importances as values. In addition, `matplotlib.pyplot` is available as `plt` and `pandas` as `pd`.

```
# Create a pd.Series of features importances
importances = pd.Series(data=rf.feature_importances_,
                        index= X_train.columns)

# Sort importances
importances_sorted = importances.sort_values()

# Draw a horizontal barplot of importances_sorted
importances_sorted.plot(kind='barh', color='lightgreen')
plt.title('Features Importances')
plt.show()
```



Apparently, `hr` and `workingday` are the most important features according to `rf`. The importances of these two features add up to more than 90%!

. . .

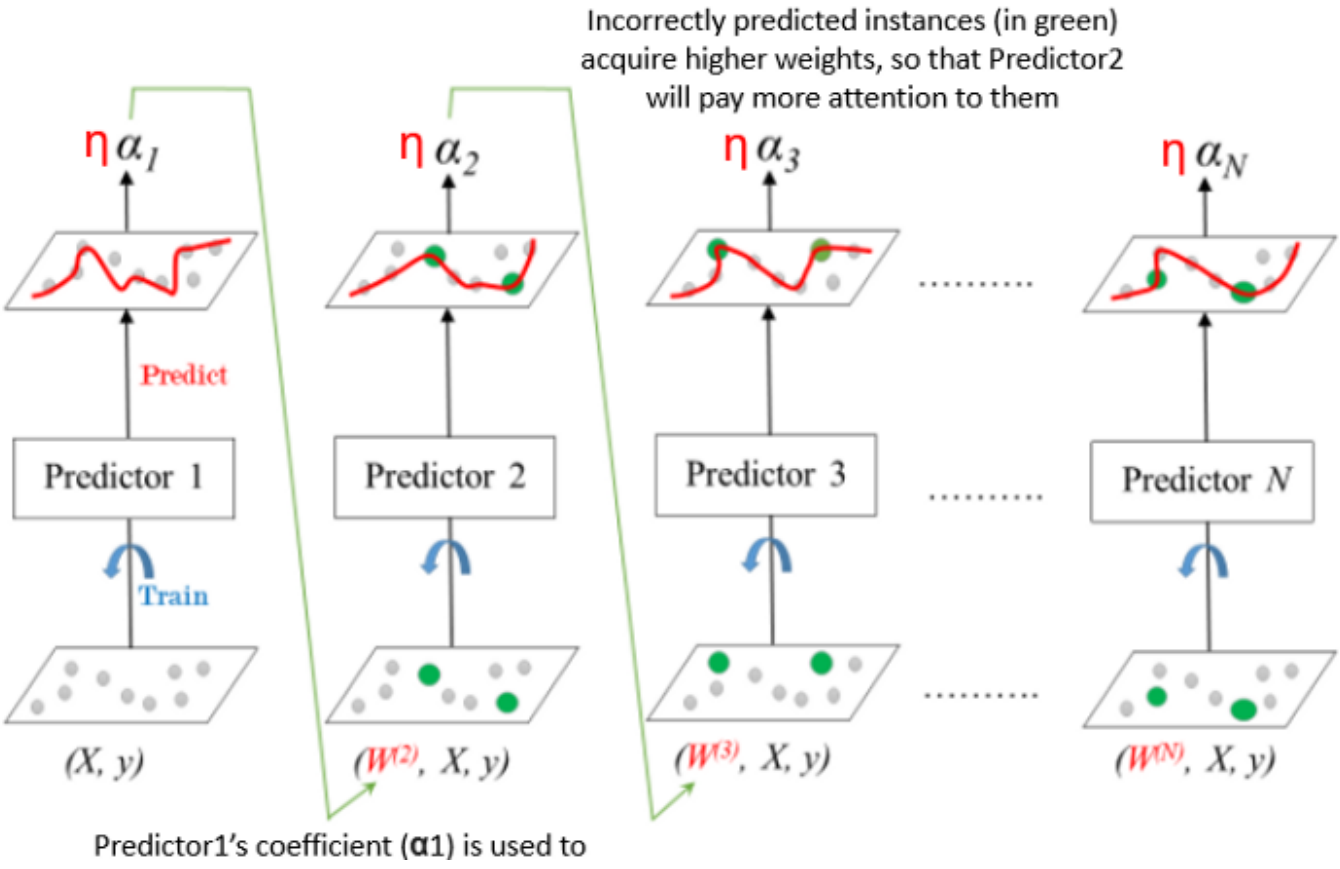
## Chapter 4. Boosting

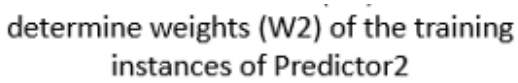
Boosting refers to an ensemble method in which several models are trained sequentially with each model learning from the errors of its predecessors. In this chapter, you'll be introduced to the two boosting methods of AdaBoost and Gradient Boosting.

### Adaboost

Boosting refers to an ensemble method in which many predictors are trained, and each predictor learns from the errors of its predecessor. Many weak learners are combined to form a strong learner.

Adaboost = **Ada**ptive **Boost**ing, each predictor pays more attention to the instances wrongly predicted by its predecessor, by constantly changing the weights of training instances.






Adaboost training

Learning rate **η** (between 0 and 1) is used to shrink the coefficient alpha of a trained predictor. A smaller learning rate should be compensated by a greater number of estimators.

Once all predictors in the ensemble are trained, the label of the unseen data can be predicted.
In classification, the final prediction is obtained by **weighted** majority voting, using the `AdaBoostClassifier` in scikit-learn.
In regression, the final prediction is the **weighted** average of the N model predictions, using the `AdaBoostRegressor` in scikit-learn.

CARTs are usually (but not necessarily) used in boosting because of their high variance.

```
# Import models and utility functions
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import train_test_split

# Set seed for reproducibility
SEED = 1

# Split data into 70% train and 30% test
X_train, X_test, y_train, y_test = train_test_split(X, y,
            test_size=0.3, stratify=y, random_state=SEED)

# Instantiate a classification-tree 'dt'
dt = DecisionTreeClassifier(max_depth=1, random_state=SEED)

# Instantiate an AdaBoost classifier 'adab_clf'
adb_clf = AdaBoostClassifier(base_estimator=dt, n_estimators=100)

# Fit 'adb_clf' to the training set
adb_clf.fit(X_train, y_train)

# Predict the test set probabilities of positive class
y_pred_proba = adb_clf.predict_proba(X_test)[:,1]
```

```
# Evaluate test-set roc_auc_score
adb_clf_roc_auc_score = roc_auc_score(y_test, y_pred_proba)

# Print adb_clf_roc_auc_score
print('ROC AUC score: {:.2f}'.format(adb_clf_roc_auc_score))
```

```
# Print adb_clf_roc_auc_score
print('ROC AUC score: {:.2f}'.format(adb_clf_roc_auc_score))
```

```
ROC AUC score: 0.99
```

Once the classifier adb_clf is trained, call .predict_proba(X_test). Extract these probabilities by slicing all the values in the second column [:,1].

### Define the AdaBoost classifier

In the following exercises you'll revisit the **Indian Liver Patient** dataset which was introduced in a previous chapter. Your task is to predict whether a patient suffers from a liver disease using 10 features including Albumin, age and gender. However, this time, you'll be training an AdaBoost ensemble to perform the classification task. In addition, given that this dataset is imbalanced, you'll be using the ROC AUC score as a metric instead of accuracy.

```
# Import DecisionTreeClassifier
from sklearn.tree import DecisionTreeClassifier

# Import AdaBoostClassifier
from sklearn.ensemble import AdaBoostClassifier

# Instantiate dt
dt = DecisionTreeClassifier(max_depth=2, random_state=1)

# Instantiate ada
ada = AdaBoostClassifier(base_estimator=dt, n_estimators=180,
random_state=1)
```

### Train the AdaBoost classifier

Now that you've instantiated the AdaBoost classifier `ada`, it's time train it. You will also predict the probabilities of obtaining the positive class in the test set. This can be done as follows:

Once the classifier `ada` is trained, call the `.predict_proba()` method by passing `X_test` as a parameter and extract these probabilities by slicing all the values in the second column: `ada.predict_proba(X_test)[:,1]`

The Indian Liver dataset is processed for you and split into 80% train and 20% test. Feature matrices `X_train` and `X_test`, as well as the arrays of labels `y_train` and `y_test` are available in your workspace. In addition, we have also loaded the instantiated model `ada` from the previous exercise.

```
# Fit ada to the training set
ada.fit(X_train, y_train)

# Compute the probabilities of obtaining the positive class
y_pred_proba = ada.predict_proba(X_test)[:,1]
```

### Evaluate the AdaBoost classifier

Now that you're done training `ada` and predicting the probabilities of obtaining the positive class in the test set, it's time to evaluate `ada`'s ROC AUC score. Recall that the ROC AUC score of a binary classifier can be determined using the `roc_auc_score()` function from `sklearn.metrics`.

The arrays `y_test` and `y_pred_proba` that you computed in the previous exercise are available in your workspace.

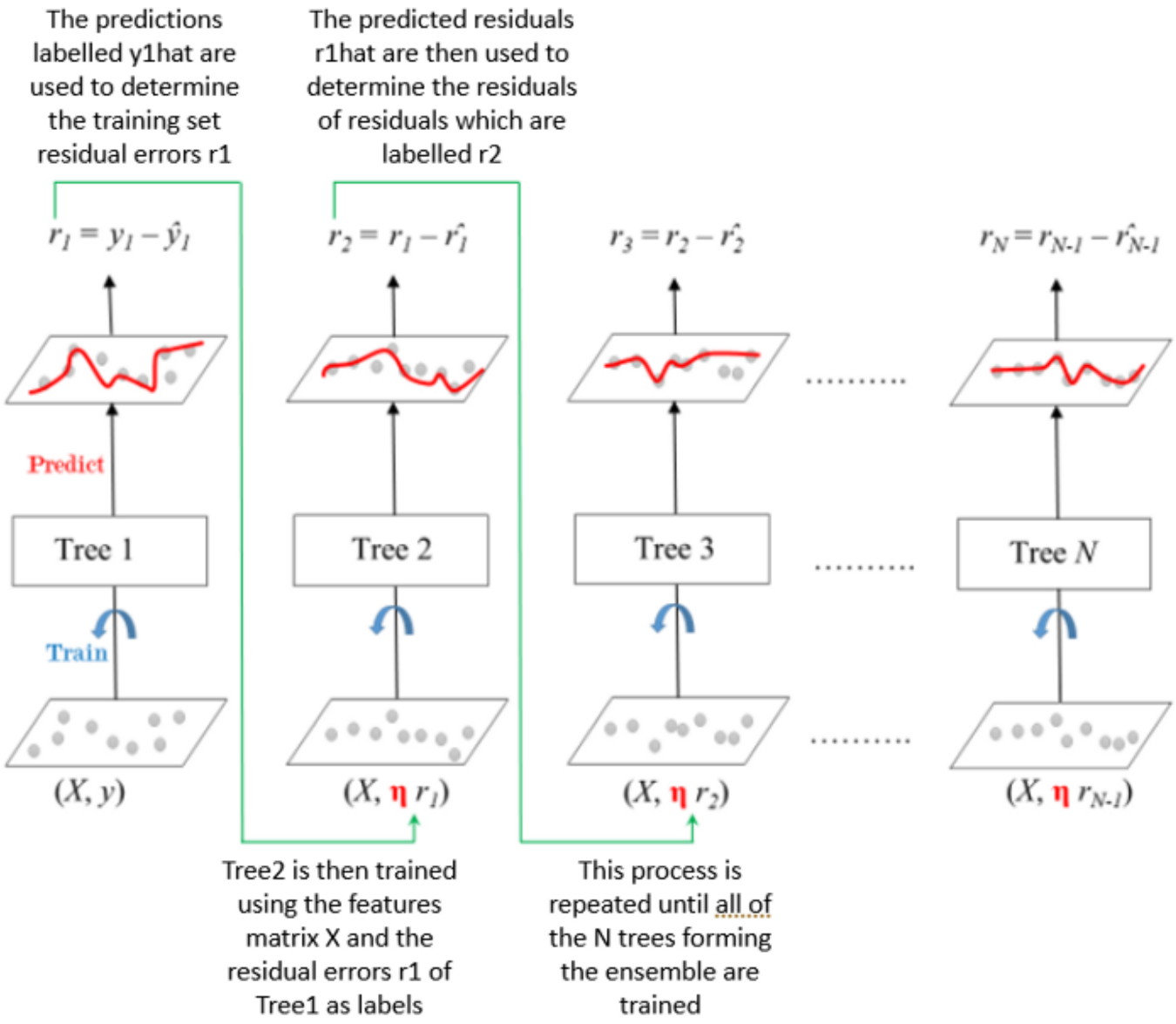
```
# Import roc_auc_score
from sklearn.metrics import roc_auc_score

# Evaluate test-set roc_auc_score
ada_roc_auc = roc_auc_score(y_test, y_pred_proba)

# Print roc_auc_score
print('ROC AUC score: {:.2f}'.format(ada_roc_auc))
```

```
<script.py> output:
    ROC AUC score: 0.71
```

This untuned AdaBoost classifier achieved a ROC AUC score of 0.71!

## Gradient Boosting (GB)

In gradient boosting, each predictor in the ensemble sequentially corrects its predecessor's error, but does not tweak the weights of the training instances. Instead, each predictor is trained using the residual errors of its predecessors as labels.




Training Gradient Boosted Trees for Regression (N trees)

Parameter **shrinkage** refers to the fact that the prediction of each tree in the ensemble is shrinked after it is multiplied by a learning rate **η** (between 0 and 1). A smaller learning rate should be compensated by a greater number of estimators.

Once all trees in the ensemble are trained, predictions can be made.
In classification, the final prediction is obtained using the `GradientBoostingClassifier` in scikit-learn.
In regression, the final prediction is obtained using the `GradientBoostingRegressor` in scikit-learn.

```
# Import models and utility functions
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error as MSE

# Set seed for reproducibility
SEED = 1

# Split dataset into 70% train and 30% test
X_train, X_test, y_train, y_test = train_test_split(X,y,
            test_size=0.3,random_state=SEED)

# Instantiate a GradientBoostingRegressor 'gbt'
gbt = GradientBoostingRegressor(n_estimators=300,
            max_depth=1, random_state=SEED)

# Fit 'gbt' to the training set
gbt.fit(X_train, y_train)

# Predict the test set labels
y_pred = gbt.predict(X_test)

# Evaluate the test set RMSE
rmse_test = MSE(y_test, y_pred)**(1/2)

# Print the test set RMSE
print('Test set RMSE: {:.2f}'.format(rmse_test))
```

```
# Print the test set RMSE
print('Test set RMSE: {:.2f}'.format(rmse_test))
```

```
Test set RMSE: 4.01
```

### Define the GB regressor

You'll now revisit the **Bike Sharing Demand** dataset that was introduced in the previous chapter. Recall that your task is to predict the bike rental demand using historical weather data from the Capital Bikeshare program in Washington, D.C.. For this purpose, you'll be using a gradient boosting regressor.

```
# Import GradientBoostingRegressor
from sklearn.ensemble import GradientBoostingRegressor

# Instantiate gb
gb = GradientBoostingRegressor(n_estimators=200,
                               max_depth=4,
                               random_state=2)
```

### Train the GB regressor

You'll now train the gradient boosting regressor `gb` that you instantiated in the previous exercise and predict test set labels.

The dataset is split into 80% train and 20% test. Feature matrices `X_train` and `X_test`, as well as the arrays `y_train` and `y_test` are available in your workspace. In addition, we have also loaded the model instance `gb` that you defined in the previous exercise.

```
# Fit gb to the training set
gb.fit(X_train, y_train)

# Predict test set labels
y_pred = gb.predict(X_test)
```

### Evaluate the GB regressor

Now that the test set predictions are available, you can use them to evaluate the test set Root Mean Squared Error (RMSE) of `gb`.

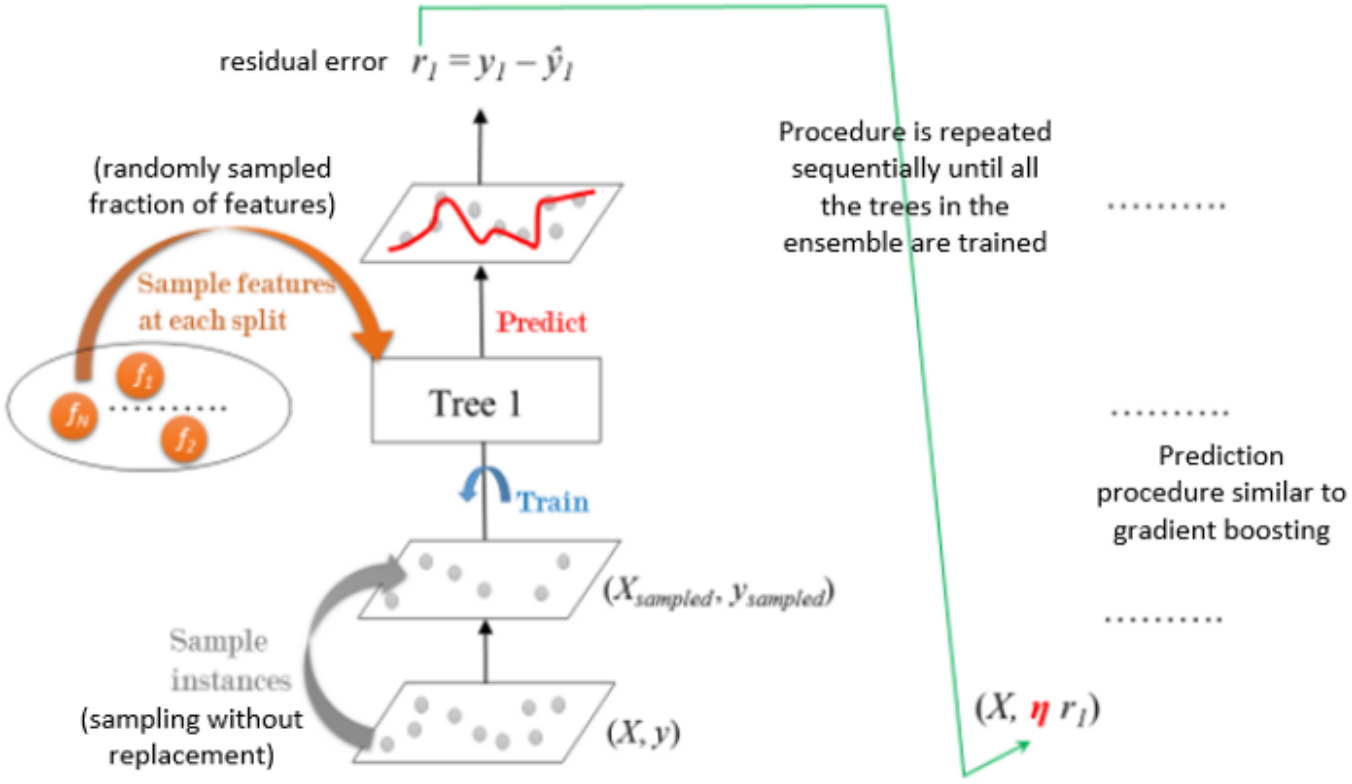
`y_test` and predictions `y_pred` are available in your workspace.

```
# Import mean_squared_error as MSE
from sklearn.metrics import mean_squared_error as MSE

# Compute MSE
mse_test = MSE(y_test, y_pred)

# Compute RMSE
rmse_test = mse_test**(0.5)

# Print RMSE
print('Test set RMSE of gb: {:.3f}'.format(rmse_test))
```

```
<script.py> output:
    Test set RMSE of gb: 52.065
```

## Stochastic Gradient Boosting (SGB)

In stochastic gradient boosting, each CART is trained on a random subset of the training data. The subset (sampled instances 40%-80% of training set) is sampled without replacement. At the level of each node, features are sampled without replacement when choosing the best split-points. As a result, this creates further diversity in the ensemble and the net effect is adding more variance to the ensemble of trees.



Training Stochastic Gradient Boosting (SGB)

```
# Import models and utility functions
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error as MSE

# Set seed for reproducibility
SEED = 1

# Split dataset into 70% train and 30% test
X_train, X_test, y_train, y_test = train_test_split(X,y,
            test_size=0.3,random_state=SEED)

# Instantiate a stochastic GradientBoostingRegressor 'sgbr'
# 0.8 = each tree samples 80% of data for training
# 0.2 = each tree uses 20% available features to do best-split
sgbr = GradientBoostingRegressor(max_depth=1,
            subsample=0.8, max_features=0.2,
            n_estimators=300, random_state=SEED)

# Fit 'sgbr' to the training set
sgbr.fit(X_train, y_train)

# Predict the test set labels
y_pred = sgbr.predict(X_test)

# Evaluate test set RMSE 'rmse_test'
rmse_test = MSE(y_test, y_pred)**(1/2)

# Print 'rmse_test'
print('Test set RMSE: {:.2f}'.format(rmse_test))
```

```
# Print 'rmse_test'
print('Test set RMSE: {:.2f}'.format(rmse_test))
```

```
Test set RMSE: 3.95
```

## Regression with SGB

As in the exercises from the previous lesson, you'll be working with the **Bike Sharing Demand** dataset. In the following set of exercises, you'll solve this bike count regression problem using stochastic gradient boosting.

```
# Import GradientBoostingRegressor
from sklearn.ensemble import GradientBoostingRegressor

# Instantiate sgbr
sgbr = GradientBoostingRegressor(max_depth=4,
            subsample=0.9,
            max_features=0.75,
            n_estimators=200,
            random_state=2)
```

### Train the SGB regressor

In this exercise, you'll train the SGBR `sgbr` instantiated in the previous exercise and predict the test set labels.

The bike sharing demand dataset is already loaded processed for you; it is split into 80% train and 20% test. The feature matrices `X_train` and `X_test`, the arrays of labels `y_train` and `y_test`, and the model instance `sgbr` that you defined in the previous exercise are available in your workspace.

```
# Fit sgbr to the training set
sgbr.fit(X_train, y_train)

# Predict test set labels
y_pred = sgbr.predict(X_test)
```

### Evaluate the SGB regressor

You have prepared the ground to determine the test set RMSE of `sgbr` which you shall evaluate in this exercise.

```
# Import mean_squared_error as MSE
from sklearn.metrics import mean_squared_error as MSE

# Compute test set MSE
mse_test = MSE(y_test, y_pred)

# Compute test set RMSE
rmse_test = mse_test**(0.5)
```

```
# Print rmse_test
print('Test set RMSE of sgbr: {:.3f}'.format(rmse_test))
```

```
<script.py> output:
    Test set RMSE of sgbr: 49.979
```

The stochastic gradient boosting regressor achieves a lower test set RMSE than the gradient boosting regressor (which was `52.065`)!

. . .

## Chapter 5. Model Tuning

The hyperparameters of a machine learning model are parameters that are not learned from data. They should be set prior to fitting the model to the training set. In this chapter, you'll learn how to tune the hyperparameters of a tree-based model using grid search cross validation.

### Tuning a CART's Hyperparameters

Model hyperparameters should be set prior to training the model, as they are not learnt from data, eg. max_depth, min_samples_leaf, etc

Hyperparameter tuning is to search for the set of optimal hyperparameters for the learning algorithm to yield an optimal model. An optimal score (highest agreement between true labels and predictions) is also obtained:
In classification: score = **accuracy**
In regression: score = $\mathbf{R^2}$

Approach to use Grid Search Cross Validation:
1. manually set a grid of discrete hyperparameter values
2. pick a metric for scoring model performance, search through the grid
3. evaluate model's CV score with each set of hyperparameter values
4. optimal hyperparameters achieve highest CV score

Grid Search suffers from the curse of dimensionality, ie, the bigger the grid, the longer it takes to find the solution.

```
## Inspecting the hyperparameters of a CART in sklearn
# Import DecisionTreeClassifier
from sklearn.tree import DecisionTreeClassifier

# Set seed to 1 for reproducibility
SEED = 1

# Instantiate a DecisionTreeClassifier 'dt'
dt = DecisionTreeClassifier(random_state=SEED)

# Print out 'dt's hyperparameters
print(dt.get_params())
```

```
{'class_weight': None,
 'criterion': 'gini',
 'max_depth': None,
 'max_features': None,
 'max_leaf_nodes': None,
 'min_impurity_decrease': 0.0,
 'min_impurity_split': None,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'presort': False,
 'random_state': 1,
 'splitter': 'best'}
```

To optimise only max_depth, max_features, min_samples_leaf

```
# Import GridSearchCV
from sklearn.model_selection import GridSearchCV

# Define the grid of hyperparameters 'params_dt'
params_dt = {
              'max_depth': [3, 4, 5, 6],
              'min_samples_leaf': [0.04, 0.06, 0.08],
              'max_features': [0.2, 0.4,0.6, 0.8]
            }

# Instantiate a 10-fold CV grid search object 'grid_dt'
grid_dt = GridSearchCV(estimator=dt, param_grid=params_dt,
            scoring='accuracy', cv=10, n_jobs=-1)
```

```
# Fit 'grid_dt' to the training data
grid_dt.fit(X_train, y_train)

# Extract best hyperparameters from 'grid_dt'
best_hyperparams = grid_dt.best_params_
print('Best hyerparameters:\n', best_hyperparams)
```

```
Best hyerparameters:
  {'max_depth': 3, 'max_features': 0.4, 'min_samples_leaf': 0.06}
```

```
# Extract best CV score from 'grid_dt'
best_CV_score = grid_dt.best_score_
print('Best CV accuracy'.format(best_CV_score))
```

```
Best CV accuracy: 0.938
```

```
# Extract best model from 'grid_dt'
best_model = grid_dt.best_estimator_

# Evaluate test set accuracy
test_acc = best_model.score(X_test, y_test)

# Print test set accuracy
print("Test set accuracy of best model: {:.3f}".format(test_acc))
```

```
Test set accuracy of best model: 0.947
```

## Tree hyperparameters

In the following exercises you'll revisit the **Indian Liver Patient** dataset which was introduced in a previous chapter.

Your task is to tune the hyperparameters of a classification tree. Given that this dataset is imbalanced, you'll be using the ROC AUC score as a metric instead of accuracy.

We have instantiated a `DecisionTreeClassifier` and assigned to `dt` with `sklearn`'s default hyperparameters. You can inspect the hyperparameters of `dt` in your console.

Which of the following is not a hyperparameter of `dt`?

- `min_impurity_decrease`
- `min_weight_fraction_leaf`
- `min_features`
- `splitter`

Answer: There is no hyperparameter named `min_features`.

### Set the tree's hyperparameter grid

In this exercise, you'll manually set the grid of hyperparameters that will be used to tune the classification tree `dt` and find the optimal classifier in the next exercise.

```
# Define params_dt
params_dt = {
              'max_depth': [2, 3, 4],
              'min_samples_leaf': [0.12, 0.14, 0.16, 0.18]
            }
```

### Search for the optimal tree

In this exercise, you'll perform grid search using 5-fold cross validation to find `dt`'s optimal hyperparameters. Note that because grid search is an exhaustive process, it may take a lot time to train the model. Here you'll only be instantiating the `GridSearchCV` object without fitting it to the training set. You can train such an object similar to any scikit-learn estimator by using the `.fit()` method: `grid_object.fit(X_train, y_train)`

An untuned classification tree `dt` as well as the dictionary `params_dt` that you defined in the previous exercise are available in your workspace.

```
# Import GridSearchCV
from sklearn.model_selection import GridSearchCV

# Instantiate grid_dt
grid_dt = GridSearchCV(estimator=dt,
                       param_grid=params_dt,
                       scoring='roc_auc',
                       cv=5,
                       n_jobs=-1)
```

### Evaluate the optimal tree

In this exercise, you'll evaluate the test set ROC AUC score of `grid_dt` 's optimal model.

In order to do so, you will first determine the probability of obtaining the positive label for each test set observation. You can use the method `predict_proba()` of an sklearn classifier to compute a 2D array containing the probabilities of the negative and positive class-labels respectively along columns.

The dataset is already loaded and processed for you (numerical features are standardized); it is split into 80% train and 20% test. `X_test` , `y_test` are available in your workspace. In addition, we have also loaded the trained `GridSearchCV` object `grid_dt` that you instantiated in the previous exercise. Note that `grid_dt` was trained as follows: `grid_dt.fit(X_train, y_train)`

```
# Import roc_auc_score from sklearn.metrics
from sklearn.metrics import roc_auc_score

# Extract the best estimator
best_model = grid_dt.best_estimator_

# Predict the test set probabilities of the positive class
y_pred_proba = best_model.predict_proba(X_test)[:,1]

# Compute test_roc_auc
test_roc_auc = roc_auc_score(y_test, y_pred_proba)

# Print test_roc_auc
print('Test set ROC AUC score: {:.3f}'.format(test_roc_auc))
```

<script.py> output:

```
Test set ROC AUC score: 0.610
```

An untuned classification-tree would achieve a ROC AUC score of `0.54` !

## Tuning a RF's Hyperparameters

Hyperparameter tuning is computationally expensive, and sometimes leads to very slight improvement.

```
## Inspecting RF Hyperparameters in sklearn
# Import RandomForestRegressor
from sklearn.ensemble import RandomForestRegressor

# Set seed for reproducibility
SEED = 1

# Instantiate a random forests regressor 'rf'
rf = RandomForestRegressor(random_state= SEED)

# Inspect rf' s hyperparameters
rf.get_params()
```

```
{'bootstrap': True,
 'criterion': 'mse',
 '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': 10,
 'n_jobs': -1,
 'oob_score': False,
 'random_state': 1,
 'verbose': 0,
 'warm_start': False}
```

To optimise only n_estimators, max_depth, min_samples_leaf, max_features

```
# Basic imports
from sklearn.metrics import mean_squared_error as MSE
from sklearn.model_selection import GridSearchCV

# Define a grid of hyperparameter 'params_rf'
params_rf = {
                'n_estimators': [300, 400, 500],
                'max_depth': [4, 6, 8],
                'min_samples_leaf': [0.1, 0.2],
                'max_features': ['log2','sqrt']
            }

# Instantiate 'grid_rf'
grid_rf = GridSearchCV(estimator=rf,param_grid=params_rf, cv=3,
        scoring='neg_mean_squared_error', verbose=1, n_jobs=-1)

# Searching for the best hyperparameters
# Fit 'grid_rf' to the training set
grid_rf.fit(X_train, y_train)
```

```
Fitting 3 folds for each of 36 candidates, totalling 108 fits
[Parallel(n_jobs=-1)]: Done  42 tasks      | elapsed:   10.0s
[Parallel(n_jobs=-1)]: Done 108 out of 108 | elapsed:   24.3s finished
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=4,
          max_features='log2', max_leaf_nodes=None,
          min_impurity_decrease=0.0, min_impurity_split=None,
          min_samples_leaf=0.1, min_samples_split=2,
          min_weight_fraction_leaf=0.0, n_estimators=400, n_jobs=1,
          oob_score=False, random_state=1, verbose=0, warm_start=False)
```

```
# Extract best hyperparameters from 'grid_rf'
best_hyperparams = grid_rf.best_params_
print('Best hyerparameters:\n', best_hyperparams)
```

```
Best hyerparameters:
       {'max_depth': 4,
        'max_features': 'log2',
        'min_samples_leaf': 0.1,
        'n_estimators': 400}
```

```
# Extract best model from 'grid_rf'
best_model = grid_rf.best_estimator_

# Predict the test set labels
y_pred = best_model.predict(X_test)

# Evaluate the test set RMSE
rmse_test = MSE(y_test, y_pred)**(1/2)

# Print the test set RMSE
print('Test set RMSE of rf: {:.2f}'.format(rmse_test))
```

```
Test set RMSE of rf: 3.89
```

The RMSE of untuned model is 3.98, so hyperparameter tuning yields slightly better results.

## Random forests hyperparameters

In the following exercises, you'll be revisiting the **Bike Sharing Demand** dataset that was introduced in a previous chapter. Recall that your task is to predict the bike rental demand using historical weather data from the Capital Bikeshare program in Washington, D.C.. For this purpose, you'll be tuning the hyperparameters of a Random Forests regressor.

We have instantiated a `RandomForestRegressor` called `rf` using `sklearn`'s default hyperparameters. You can inspect the hyperparameters of `rf` in your console.

Which of the following is not a hyperparameter of `rf`?

- `min_weight_fraction_leaf`
- `criterion`
- `learning_rate`
- `warm_start`

Answer: There is no hyperparameter named `learning_rate`.

### Set the hyperparameter grid of RF

In this exercise, you'll manually set the grid of hyperparameters that will be used to tune `rf`'s hyperparameters and find the optimal regressor. For this purpose, you will be constructing a grid of hyperparameters and tune the number of estimators, the maximum number of features used when splitting each node and the minimum number of samples (or fraction) per leaf.

```
# Define the dictionary 'params_rf'
params_rf = {
                'n_estimators': [100, 350, 500],
                'min_samples_leaf': [2, 10, 30],
                'max_features': ['log2', 'auto', 'sqrt']
            }
```

### Search for the optimal forest

In this exercise, you'll perform grid search using 3-fold cross validation to find `rf`'s optimal hyperparameters. To evaluate each model in the grid, you'll be using the **negative mean squared error** metric.

Note that because grid search is an exhaustive search process, it may take a lot time to train the model. Here you'll only be instantiating the `GridSearchCV` object without fitting it to the training set. You can train such an object similar to any scikit-learn estimator by using the `.fit()` method: `grid_object.fit(X_train, y_train)`

The untuned random forests regressor model `rf` as well as the dictionary `params_rf` that you defined in the previous exercise are available in your workspace.

```
# Import GridSearchCV
from sklearn.model_selection import GridSearchCV

# Instantiate grid_rf
grid_rf = GridSearchCV(estimator=rf,
                       param_grid=params_rf,
                       scoring='neg_mean_squared_error',
                       cv=3,
```

```
            verbose=1,
            n_jobs=-1)
```

### Evaluate the optimal forest

In this last exercise of the course, you'll evaluate the test set RMSE of `grid_rf` 's optimal model.

The dataset is already loaded and processed for you and is split into 80% train and 20% test. In your environment are available `X_test` , `y_test` and the function `mean_squared_error` from `sklearn.metrics` under the alias `MSE` . In addition, we have also loaded the trained `GridSearchCV` object `grid_rf` that you instantiated in the previous exercise. Note that `grid_rf` was trained as follows: `grid_rf.fit(X_train, y_train)`

```
# Import mean_squared_error from sklearn.metrics as MSE
from sklearn.metrics import mean_squared_error as MSE

# Extract the best estimator
best_model = grid_rf.best_estimator_

# Predict test set labels
y_pred = best_model.predict(X_test)

# Compute rmse_test
rmse_test = MSE(y_test, y_pred)**(1/2)

# Print rmse_test
print('Test RMSE of best model: {:.3f}'.format(rmse_test))
```

```
<script.py> output:
    Test RMSE of best model: 50.569
```

. . .

## Congratulations!

Let's recap the topics covered in this course:

Chapter 1: Decision-Tree Learning, applying CART algorithm to train decision trees for classification/regression problems.

Chapter 2: Generalization Error of a supervised learning model, to diagnose underfitting and overfitting using Cross-Validation. Ensembling can produce better results than individual decision trees.

Chapter 3: Bagging, applied randomization through bootstrapping and constructed a diverse set of trees in an ensemble through bagging. Random Forests introduces further randomization by sampling features at the level of each node in each tree forming the ensemble.

Chapter 4. AdaBoost and Gradient-Boosting, an ensemble method in which predictors are trained sequentially and each predictor tries to correct the errors made by its predecessors. Subsampling instances and features can lead to a better performance through Stochastic Gradient Boosting.

Chapter 5. Model hyperparameter tuning through Grid Search CV.

. . .

Happy learning!

Thanks to TDS Team (hide).

Machine Learning Decision Making Python Supervised Learning Classification