Machine Learning with Tree-Based Models

Tutorial on the fundamental concepts in supervised machine learning

[Black Raven (James Ng)](https://medium.com/@jnyh?source=post_page-----51261c4eaae6----------------------)

[Black Raven (James Ng)](https://medium.com/@jnyh?source=post_page-----51261c4eaae6----------------------)

[Aug 10](https://medium.com/ai-in-plain-english/machine-learning-with-tree-based-models-51261c4eaae6?source=post_page-----51261c4eaae6----------------------) · 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](https://github.com/JNYH/DataCamp_Machine_Learning_with_Tree-Based_Models):

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](https://unsplash.com/@leliejens?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

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)



**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**](https://www.kaggle.com/uciml/breast-cancer-wisconsin-data) 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])

Image for post

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))

Image for post

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.

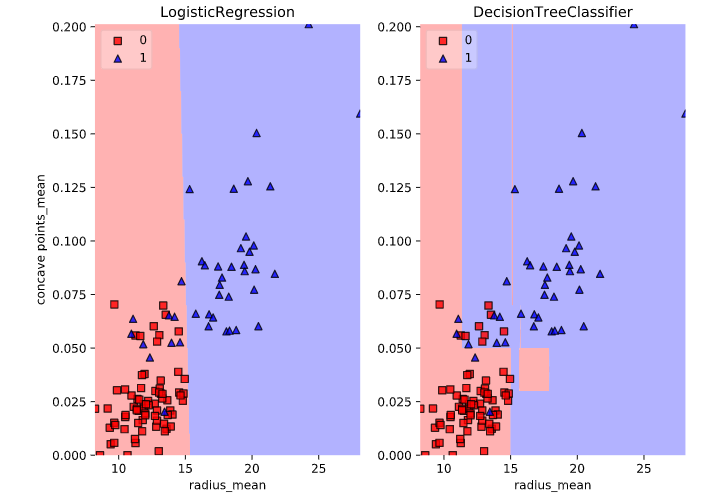
# Import LogisticRegression from sklearn.linear\_model  
from sklearn.linear\_model import LogisticRegression

# 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]

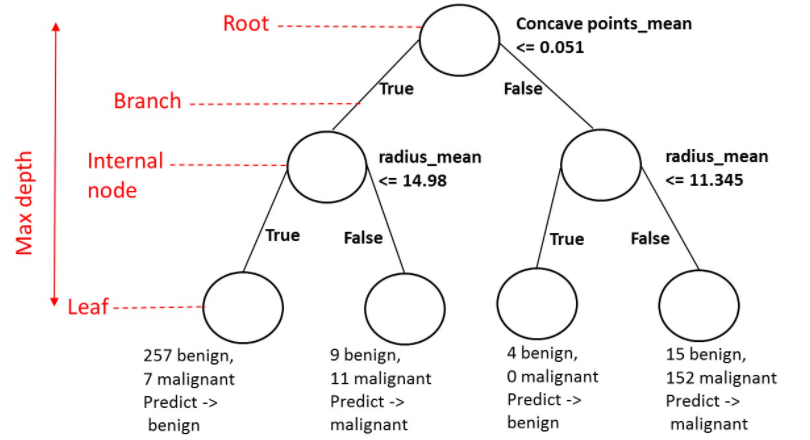
# Review the decision regions of the two classifiers  
plot\_labeled\_decision\_regions(X\_test, y\_test, clfs)



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.

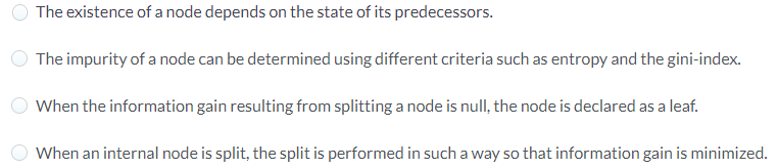
**Classification tree Learning**

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?



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.

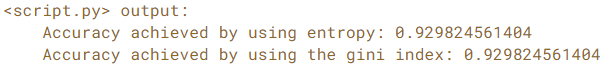
# Import accuracy\_score from sklearn.metrics  
from sklearn.metrics import accuracy\_score

# 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)



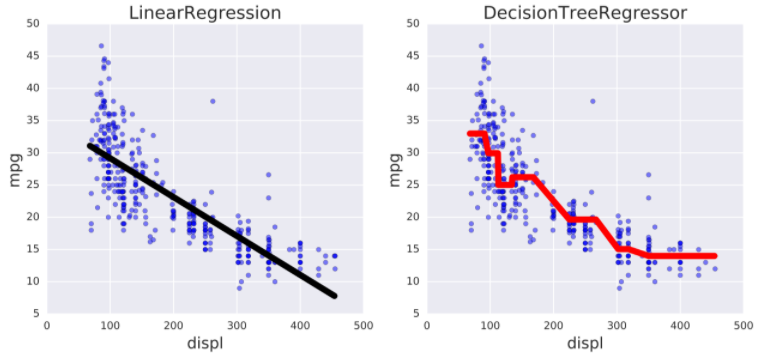
Notice how the two models achieve exactly the same accuracy. Most of the time, the gini index and entropy lead to the same results. The **gini index**is slightly faster to compute and is the **default**criterion used in the DecisionTreeClassifier model of scikit-learn.

**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.





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**](https://www.kaggle.com/uciml/autompg-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))

Image for post

**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.

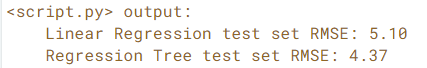
# Predict test set labels   
y\_pred\_lr = lr.predict(X\_test)

# 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))

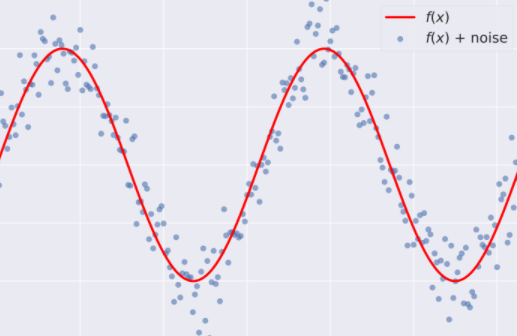


**Chapter 2. The Bias-Variance Tradeoff**

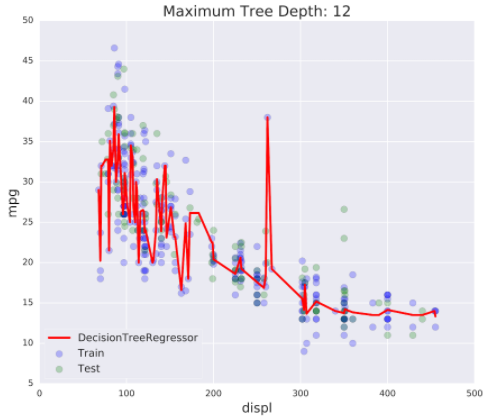
The bias-variance tradeoff is one of the fundamental concepts in supervised machine learning. In this chapter, you’ll understand how to diagnose the problems of overfitting and underfitting. You’ll also be introduced to the concept of ensembling where the predictions of several models are aggregated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

**Generalization Error**

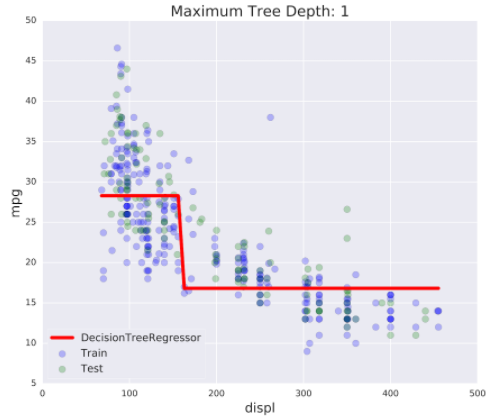
In supervised learning, the assumption is made that there is a mapping f between features and labels: y = f(x)



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.



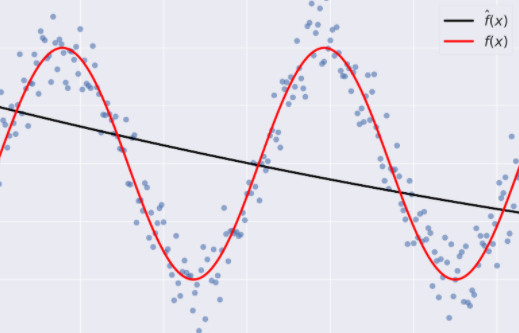
**Overfitting**= low training set error, high test set error



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

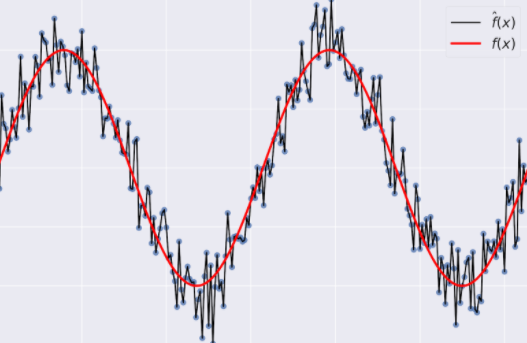
**Generalization Error** (GE) shows how well it generalizes on unseen data.  
GE of a model = Bias + Variance + Noise

**Bias**= how different are ‘f\_approx’ and ‘f’. High bias models lead to underftting.

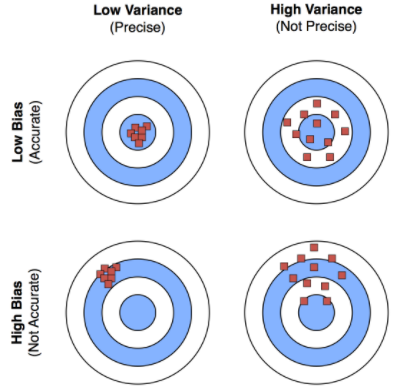


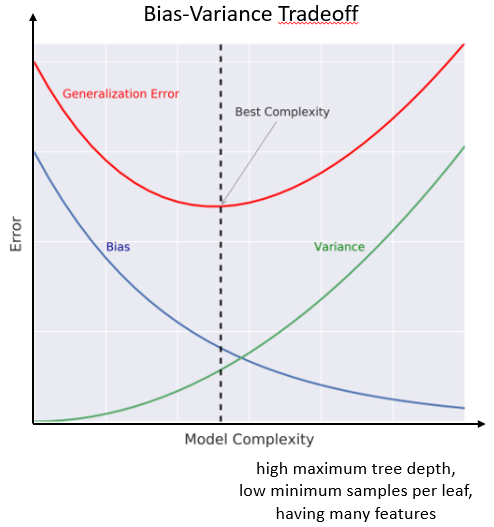
**High bias** model (f\_approx) in black, true function (f) in red

**Variance**= how the ‘f\_approx’ is inconsistent over different training sets. High variance models lead to overfitting.



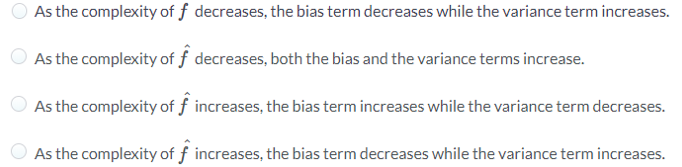
**High variance** model (f\_approx) in black, true function (f) in red





**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?



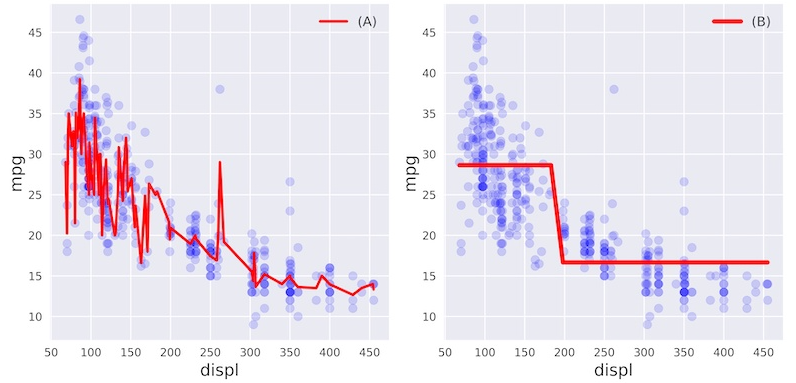
Answer: As the complexity of **f^** increases, the bias term decreases while the variance term increases.

**Overfitting and underfitting**

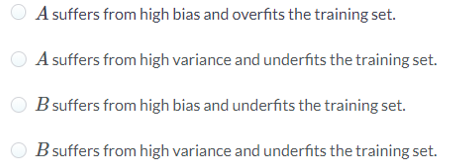
In this exercise, you’ll visually diagnose whether a model is overfitting or underfitting the training set.

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.



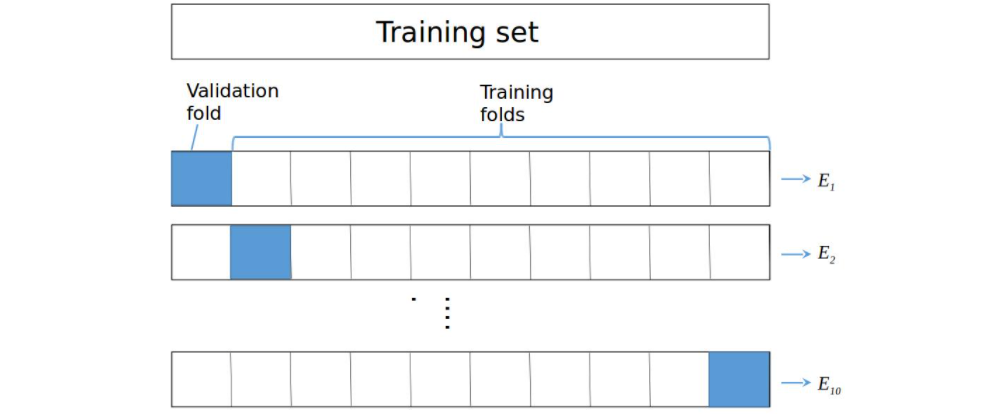
Which of the following statements is true?



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.

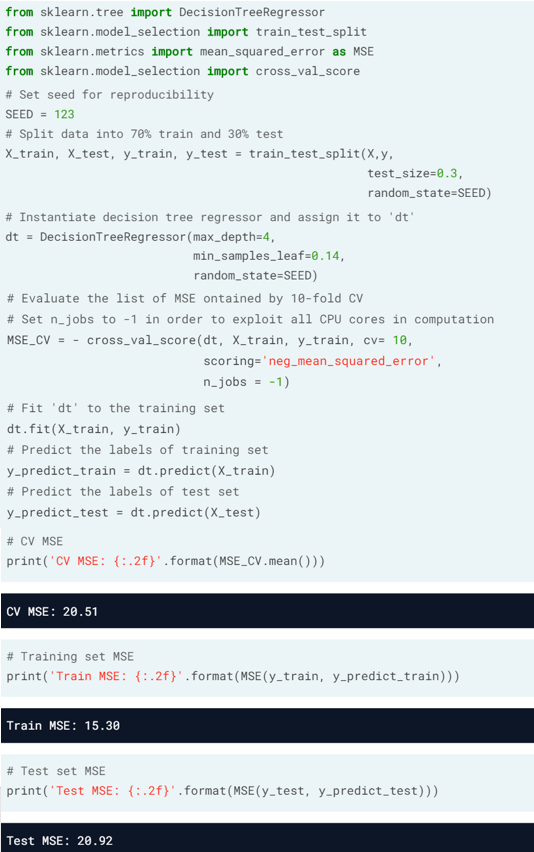
**Diagnose bias and variance problems**

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.



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))

Image for post

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))

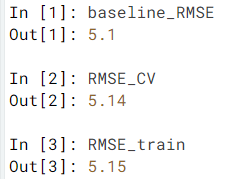
Image for post

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

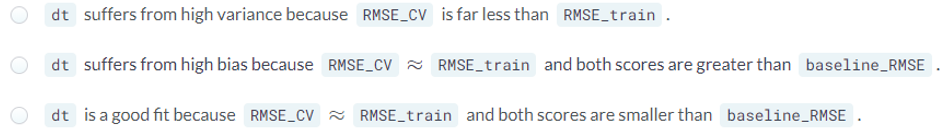
**High bias or high variance?**

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'.



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

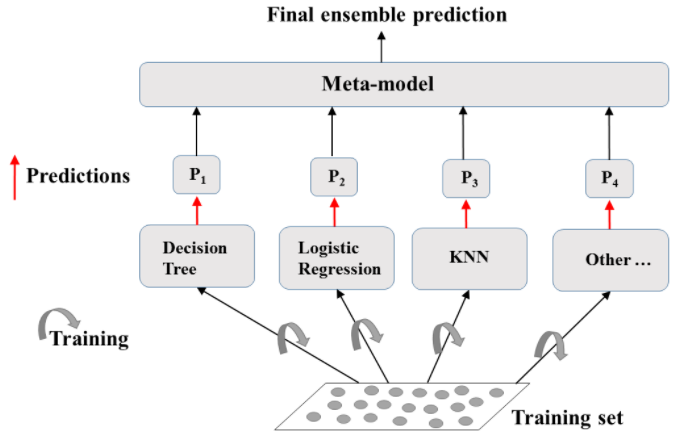


Answer: dt suffers from high bias because RMSE\_CV **≈** RMSE\_train and both scores are greater than baseline\_RMSE. The model is underfitting the training set as the model is too constrained to capture the nonlinear dependencies between features and labels.

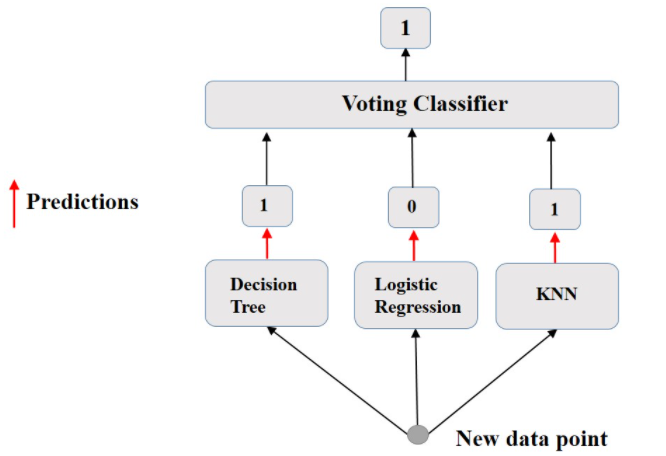
**Ensemble Learning**

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.



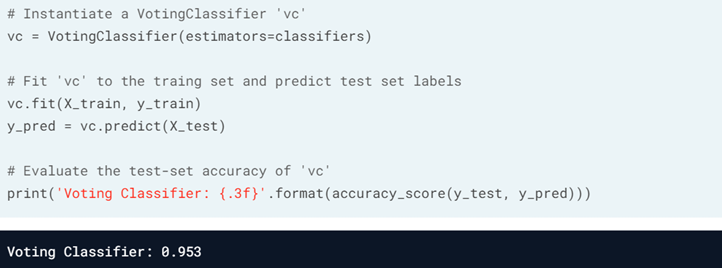
Ensemble Learning



Voting Classifier



Results from individual classifiers



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**](https://www.kaggle.com/jeevannagaraj/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)

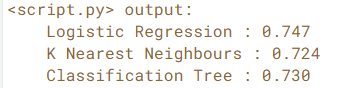
# Define the list classifiers  
classifiers = [('Logistic Regression', lr), ('K Nearest Neighbours', knn), ('Classification Tree', dt)]

**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))



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))

Image for post

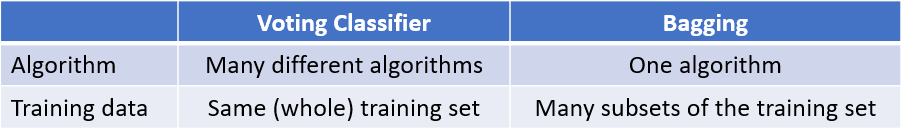
Note: The voting classifier achieves a test set accuracy of 75.3%. This value is greater than that previously achieved by LogisticRegression (74.7%).

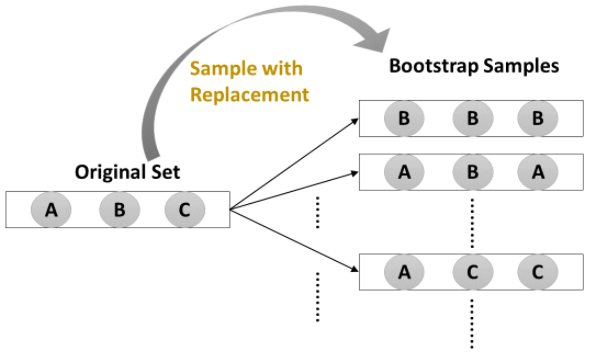
**Chapter 3. Bagging and Random Forests**

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.

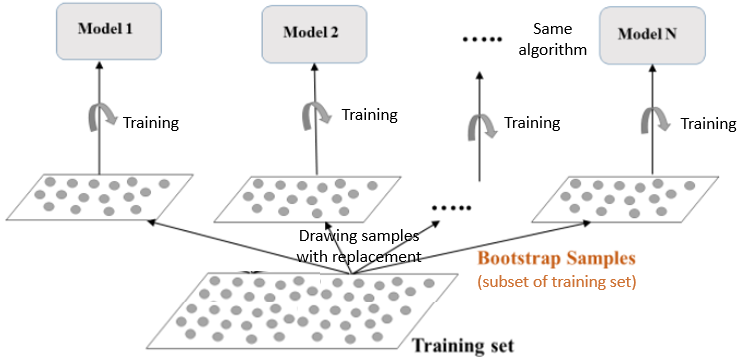
**Bagging**

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.





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))

Image for post

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**](https://www.kaggle.com/uciml/indian-liver-patient-records) 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))

Image for post

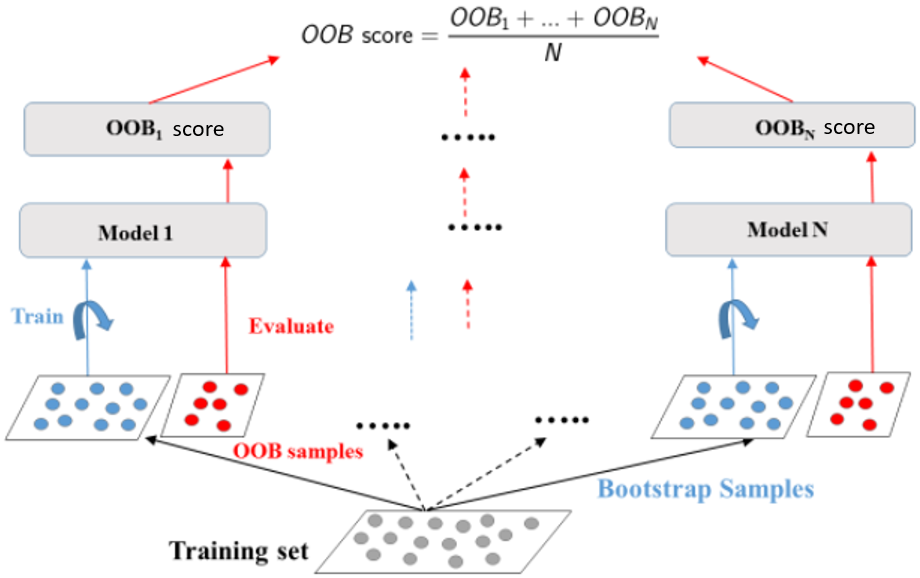
# Cross check dt performance  
dt.fit(X\_train, y\_train)  
y\_pred = dt.predict(X\_test)  
acc\_test = accuracy\_score(y\_test, y\_pred)  
print('Test set accuracy of dt: {:.2f}'.format(acc\_test))

Image for post

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 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)



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\_

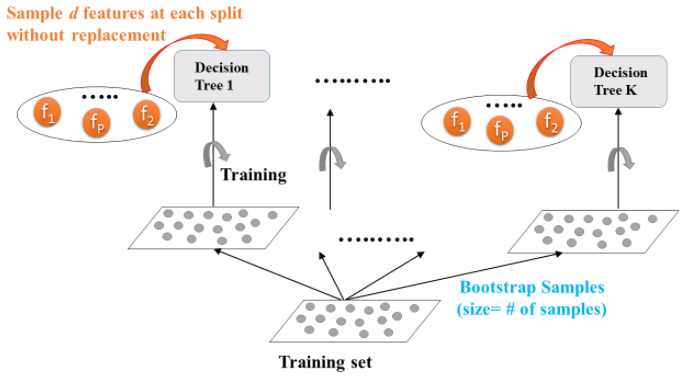
# Print acc\_test and acc\_oob  
print('Test set accuracy: {:.3f}, OOB accuracy: {:.3f}'.format(acc\_test, acc\_oob))

Image for post

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)

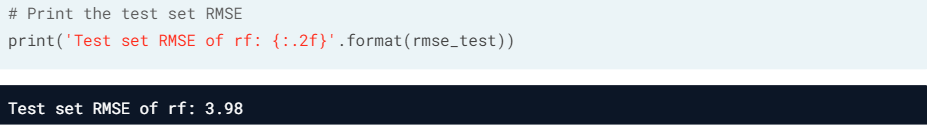
# Instantiate a random forests regressor 'rf' 400 estimators  
rf = RandomForestRegressor(n\_estimators=400,   
 min\_samples\_leaf=0.12, random\_state=SEED)

# 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))



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

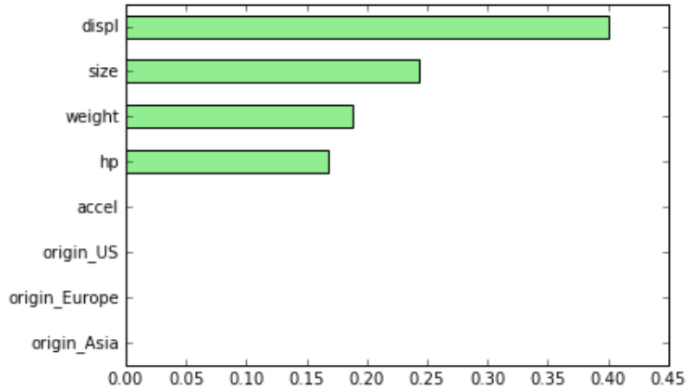
When a tree based method is trained, the predictive power of a feature (feature importance) can be assessed, my measuring how much the tree nodes use a particular feature to reduce impurity.  
Feature importance is calculated as a percentage indicating the weight of that feature in training and prediction.

# 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**](https://www.kaggle.com/c/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))

Image for post

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.

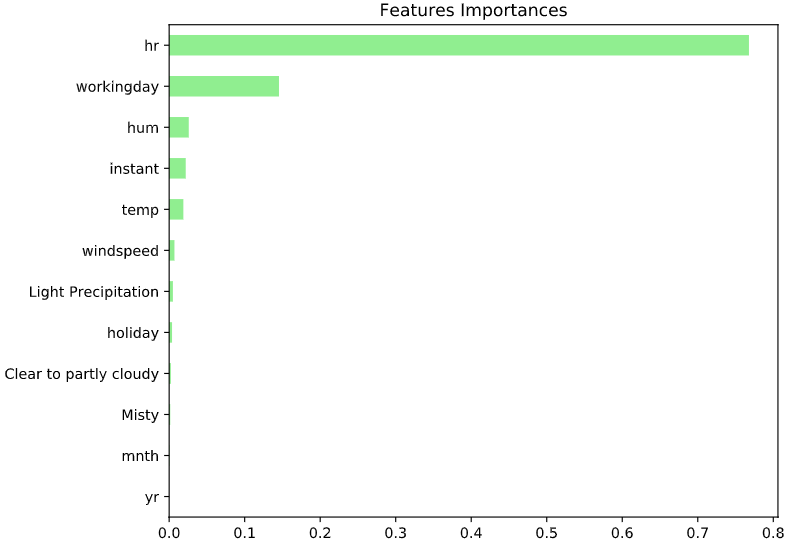
For this purpose, you’ll draw a horizontal barplot of the feature importance as assessed by rf. Fortunately, this can be done easily thanks to plotting capabilities of pandas.

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%!

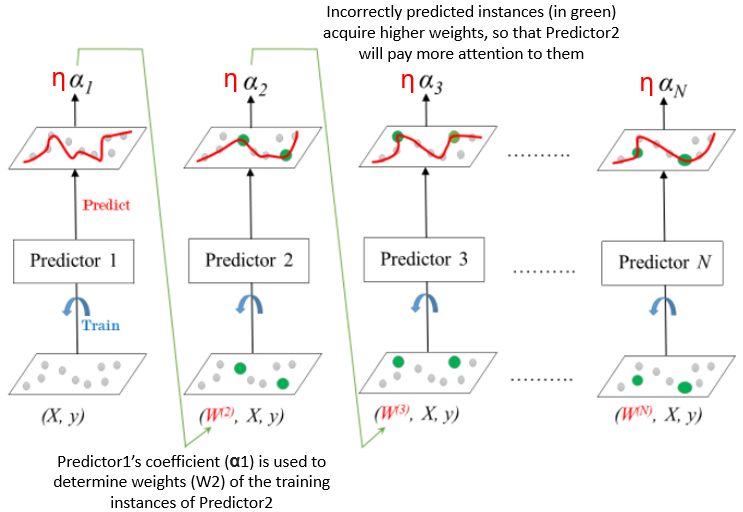
**Chapter 4. Boosting**

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)

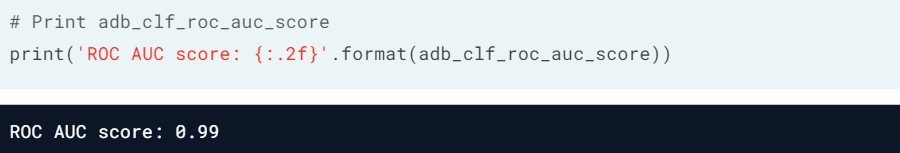
# Instantiate an AdaBoost classifier 'adab\_clf'  
adb\_clf = AdaBoostClassifier(base\_estimator=dt, n\_estimators=100)

# 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))



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**](https://www.kaggle.com/uciml/indian-liver-patient-records) 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)

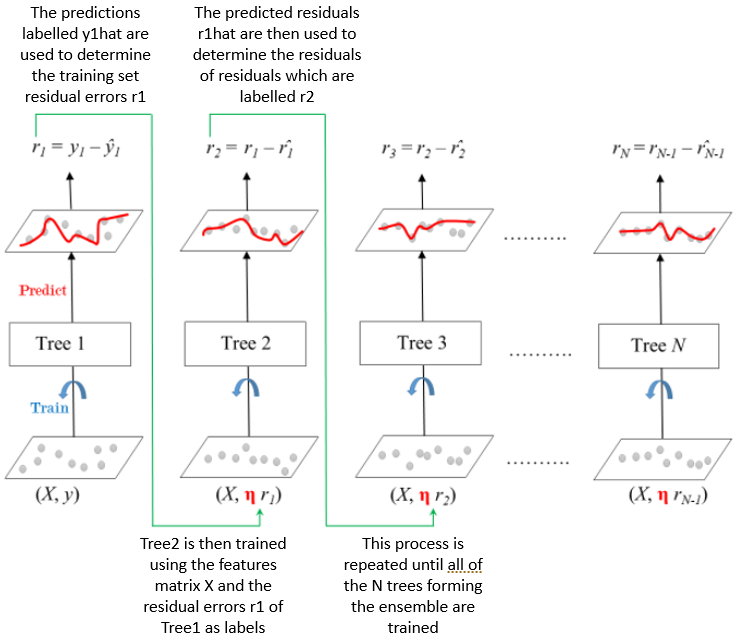
# Print roc\_auc\_score  
print('ROC AUC score: {:.2f}'.format(ada\_roc\_auc))

Image for post

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))



**Define the GB regressor**

You’ll now revisit the [**Bike Sharing Demand**](https://www.kaggle.com/c/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)

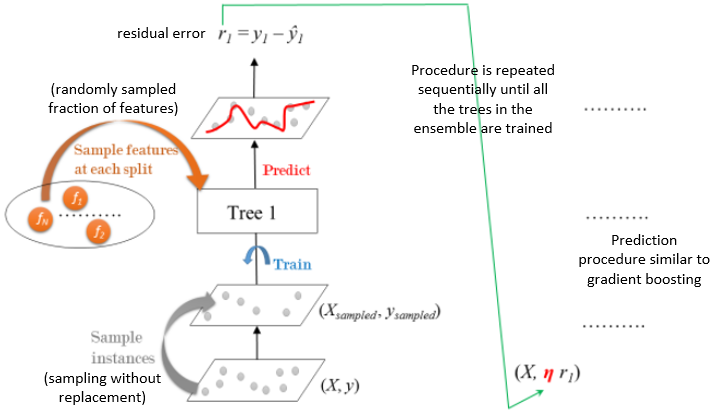
# Compute RMSE  
rmse\_test = mse\_test\*\*(0.5)

# Print RMSE  
print('Test set RMSE of gb: {:.3f}'.format(rmse\_test))

Image for post

**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))



**Regression with SGB**

As in the exercises from the previous lesson, you’ll be working with the [**Bike Sharing Demand**](https://www.kaggle.com/c/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))

Image for post

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 = **R²**

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

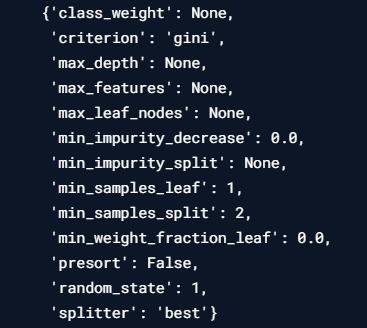
Grid Search suffers from the curse of dimensionality, ie, the bigger the grid, the longer it takes to find the solution.

## 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())



To optimise only max\_depth, max\_features, min\_samples\_leaf

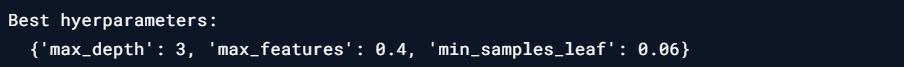
# Import GridSearchCV  
from sklearn.model\_selection import GridSearchCV

# 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)



# Extract best CV score from 'grid\_dt'  
best\_CV\_score = grid\_dt.best\_score\_  
print('Best CV accuracy'.format(best\_CV\_score))

Image for post

# 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))

Image for post

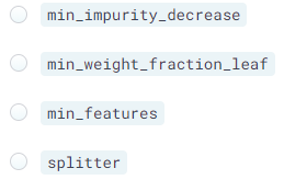
**Tree hyperparameters**

In the following exercises you’ll revisit the [**Indian Liver Patient**](https://www.kaggle.com/uciml/indian-liver-patient-records) 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?



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 methodpredict\_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))

Image for post

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

**Tuning a RF’s Hyperparameters**

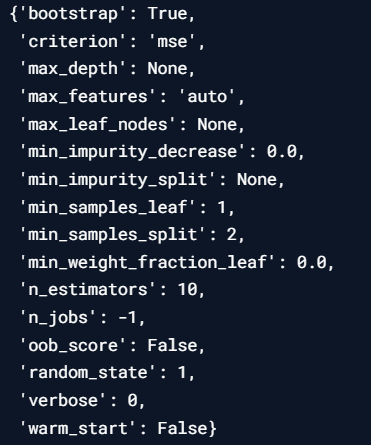
Hyperparameter tuning is computationally expensive, and sometimes leads to very slight improvement.

## 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()



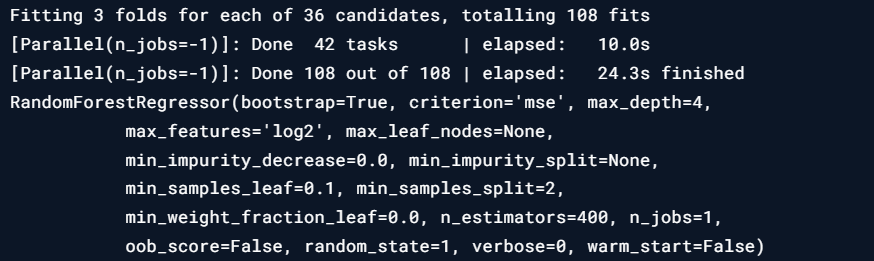
To optimise only n\_estimators, max\_depth, min\_samples\_leaf, max\_features

# Basic imports  
from sklearn.metrics import mean\_squared\_error as MSE  
from sklearn.model\_selection import GridSearchCV

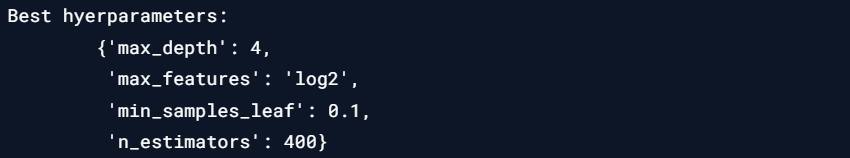
# Define a grid of hyperparameter 'params\_rf'  
params\_rf = {  
 'n\_estimators': [300, 400, 500],  
 'max\_depth': [4, 6, 8],  
 'min\_samples\_leaf': [0.1, 0.2],  
 'max\_features': ['log2','sqrt']  
 }

# 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)



# Extract best hyperparameters from 'grid\_rf'  
best\_hyperparams = grid\_rf.best\_params\_  
print('Best hyerparameters:\n', best\_hyperparams)



# 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))

Image for post

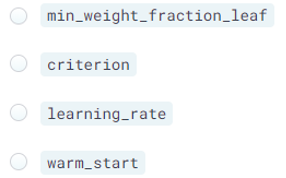
The RMSE of untuned model is 3.98, so hyperparameter tuning yields slightly better results.

**Random forests hyperparameters**

In the following exercises, you’ll be revisiting the [**Bike Sharing Demand**](https://www.kaggle.com/c/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?



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**](http://scikit-learn.org/stable/modules/model_evaluation.html) 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))

Image for post

**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!