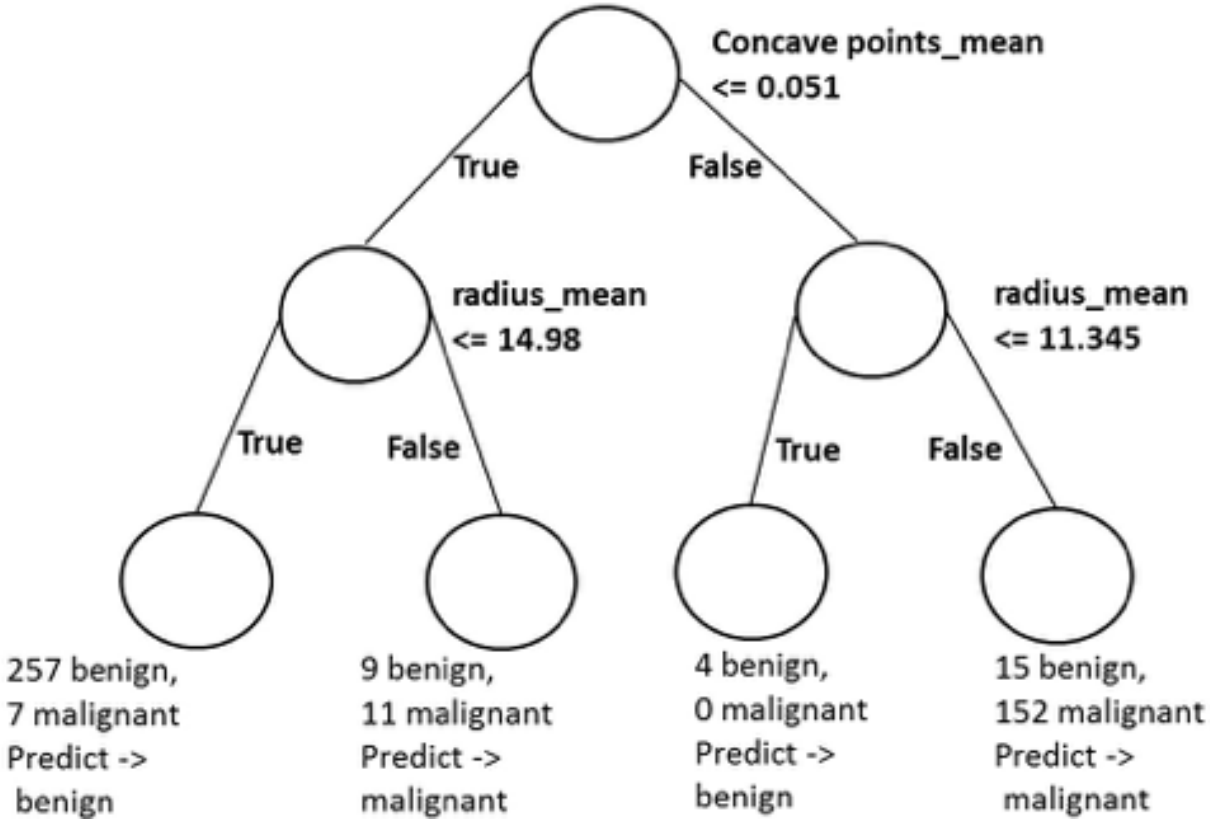
Decision tree for classification
Course by datacamp

Classification-tree
a sequence of if-else questions about individual features
with the objective of infering class labels
trees are able to capture non-linear relationships between features and labels
trees don’t require the features to be on the same scale through standardizaion for example
example - Breast Cancer dataset

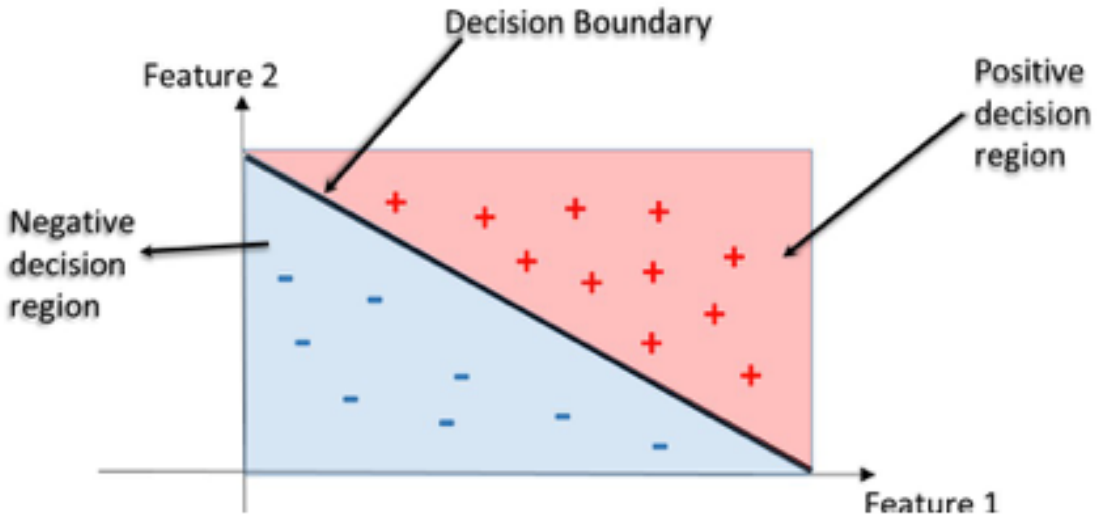


the tree learns a sequence of if-else questions with each question involving one feature and one split-point
the instance keeps traversing the internal branches until it reaches the end
the label of the instance is then predicted to be that of the prevailing class at the end
‘maximum depth’ is the maximum number of branches separating the top from the extreme-end

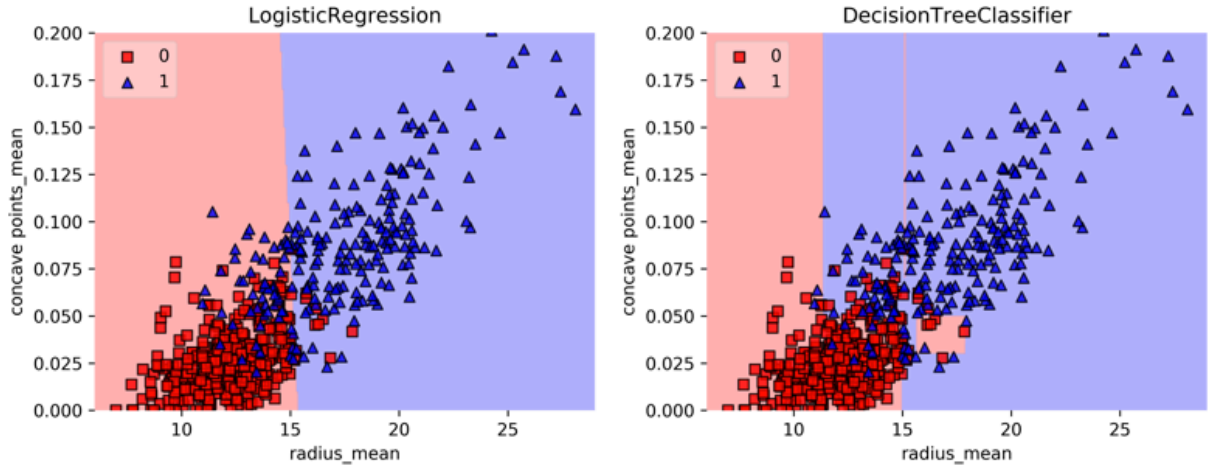
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
```

```
from sklearn.metrics import accuracy_score
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y,
random_stat=1)
#'stratify' argument to y in order for the train and test sets to have the same
proportion of class labels as the unsplit dataset
#instantiate DecisionTreeClassifier with 'max_depth' to 2 (again represents the
branches between top and bottom
dt = DecisionTreeClassifier(max_depth=2, random_state=1)
#fit to the trainind set
dt.fit(X_train, y_train)
#predict the test set accuracy
accuracy_score(y_test, y_pred)
```

Let's understand this more
to understand the tree's predictions more concretely, see how it classifies instances in the feature-space
Decision Regions
a classification-model divides the feature-space into regions where all instances in one region are assigned to only one class-label
decision regions are separated by surfaces called decision-boundaries

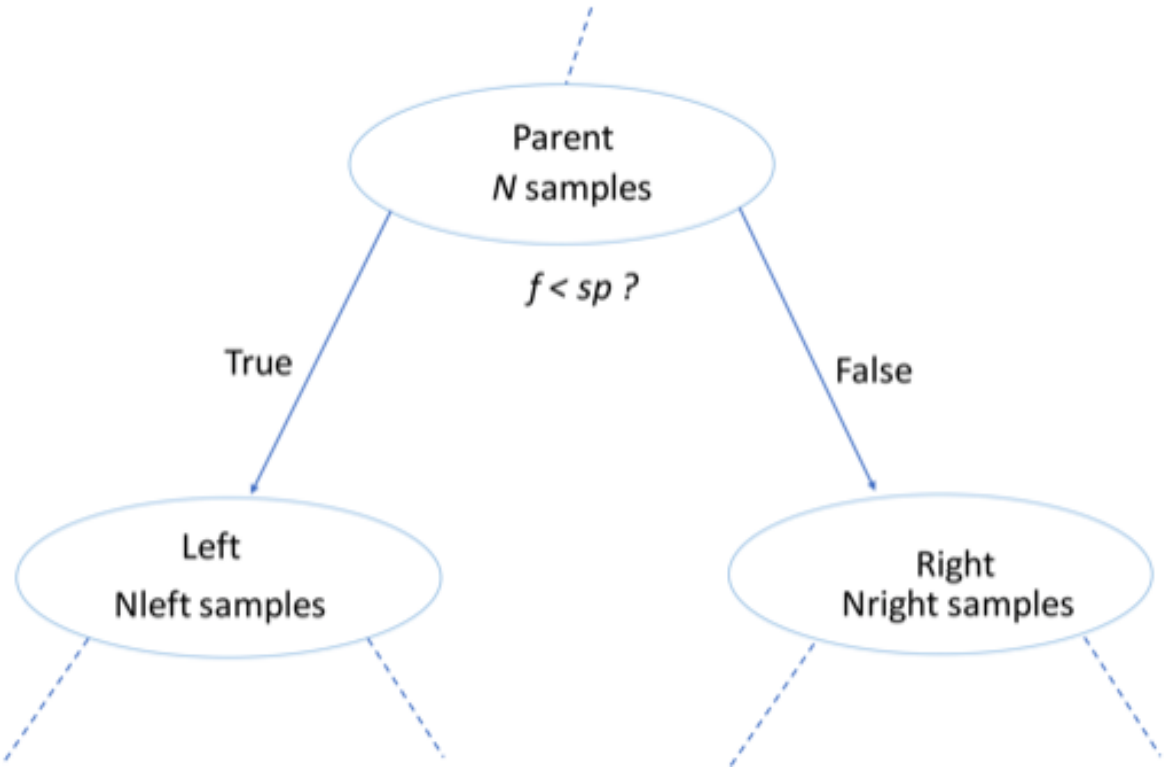


DTs produce rectangular decision-regions in the feature space
this happens because at each split made by the tree only one feature is involved



How does a classifcation tree learn?
decision tree > data structure consisting of a hierarchy of nodes (individual units)
node > question or prediction
three kinds of nodes:
root - no parent node, question giving rise to two children nodes (the start)
internal node - one parent node, question giving rise to two children nodes
leaf - one parent node, no children nodes (this is the prediction)
*recall that when a classification tree is trained on a labeled dataset, the tree learns patterns from the features in such a way to produce the purest leafs (ie one class-label is predominant)
but how?

'Information Gain (IG)'
nodes of a classification tree are grown recursively
recursive describes a process or function the repeats or applies itself in a self-referential manner
a technique where a function calls itself to solve a smaller instance of the problem
the function continues to call on itself on progressively smaller subproblems until it reaches a base case that does not require further recursion
'self-referential' referring back to itself
the obtention of an internal node or a leaf depends on the state of its predecessors
'obtention' the act of acquiring or obtaining
at each node, a tree asks a question involving one feature (f) and a split-point (sp)
how does it know which feature and which split-point to pick?
by maximizing information gain
the tree considers that every node contains information and aims at maximizing the Information Gain obtained after each split



More details on Inforamation Gain (IG)
Information gain is a concept used in the field of machine learning and decision trees to measure the effectiveness of a particular attribute in classifying or partitioning data. It is commonly used in the context of feature selection and attribute evaluation.

In decision tree algorithms, such as ID3 (Iterative Dichotomiser 3) and C4.5, information gain is used to determine the order in which attributes (features) should be selected to split the data into branches, resulting in a decision tree that best separates different classes or categories.

The main idea behind information gain is to quantify how much knowledge or reduction in uncertainty about the class labels is gained by splitting the data based on a particular attribute. The more information we gain by using an attribute to split the data, the more important that attribute is for the classification task.

Information gain is calculated based on the concept of entropy, which measures the impurity or disorder in a dataset. Entropy helps us understand how well a particular attribute divides the data into distinct classes. The formula for information gain is as follows:

Information Gain = Entropy(parent) - Weighted Average of Entropy(children)

Here, "Entropy(parent)" represents the entropy of the original dataset before the

split, and "Entropy(children)" is the weighted average of the entropy of the datasets after the split.

Higher information gain values indicate that a split based on the attribute will lead to more homogeneity within the resulting subsets, making the decision tree more effective in classifying the data.

In summary, information gain is a technique used in decision trees to select the best attributes for splitting data, optimizing the tree's ability to classify and make accurate predictions. It is a fundamental concept in machine learning algorithms that rely on decision trees and feature selection.

More on the entropy of information theory
Information Entropy:
In information theory, entropy represents the uncertainty or average amount of information in a random variable or a probability distribution. It is a measure of the "surprise" or "information content" of an event. If an event has a high probability of occurring, its information entropy is low because it carries little surprise. Conversely, if an event has a low probability, its information entropy is high because it carries more surprise or unexpectedness.

The formula for calculating information entropy of a discrete random variable X with n possible outcomes {x1, x2, ..., xn}, each with probability P(xi), is as follows:

Entropy(X) = - Σ [P(xi) * log2(P(xi))], for all i from 1 to n

Information entropy is widely used in data compression, data coding, and data transmission. It helps in quantifying the efficiency of encoding information and is a fundamental concept in the design of lossless data compression algorithms, such as Huffman coding.

In summary, entropy is a measure of disorder or uncertainty, and its interpretation depends on the context in which it is applied—either in the realm of thermodynamics, describing the randomness of a physical system, or in information theory, characterizing the uncertainty or surprise in a set of data.

$$IG(\underbrace{f}_{feature}, \underbrace{sp}_{split-point}) = I(parent) - \left(\frac{N_{left}}{N} \; I(left) + \frac{N_{right}}{N} \; I(right) \right)$$

different types of criteria can be used to measure the impurity of a node
examples > entropy, gini index, and more

*can be picked as an argument within the DecisionTreeClassifer
‘criterion’ argument
dt = DecisionTreeClassifier(criterion=‘gini’)

Classification-Tree Learning
‘constrained’ and ‘unconstrained’ trees
constrained tree if a max_depth is determined
unconstrained tree, node formation will continue until information gain is null
ie. if IG(node) = 0 declare the node a leaf
if constrained, leafs will be declared at max depth even if IG(node) is not equal to 0

Decision-tree for Regression
recall that in regression, the target variable is continuous (ie the output of your model is a real value)
example - auto-mpg dataset (6 features to determine the continuous target variable mpg
predict mpg off feature displ (corresponds to the displacment of a car)
a scatter plot of x=displ and y=mpg shows a negative non-linear trend
a linear model will not be able to capture this trend
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error as MSE
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=3)
#instantiate DecisionTreeRegressor with argument min_samples_leaf
#min_samples_leaf imposes a stopping condition in which each leaf has to contain at least 10% of the training data
dt = DecisionTreeRegressor(max_depth=4, min_samples_leaf=0.1, random_state=3)
dt.fit(X_train, y_train)
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)

Information criterion for regression-tree

$$I(\text{node}) = \underbrace{\text{MSE(node)}}_{mean-squared-error} = \frac{1}{N_{node}} \sum_{i \in node} \left(y^{(i)} - \hat{y}_{node}\right)^2$$

$$\underbrace{\hat{y}_{node}}_{mean-target-value} = \frac{1}{N_{node}} \sum_{i \in node} y^{(i)}$$

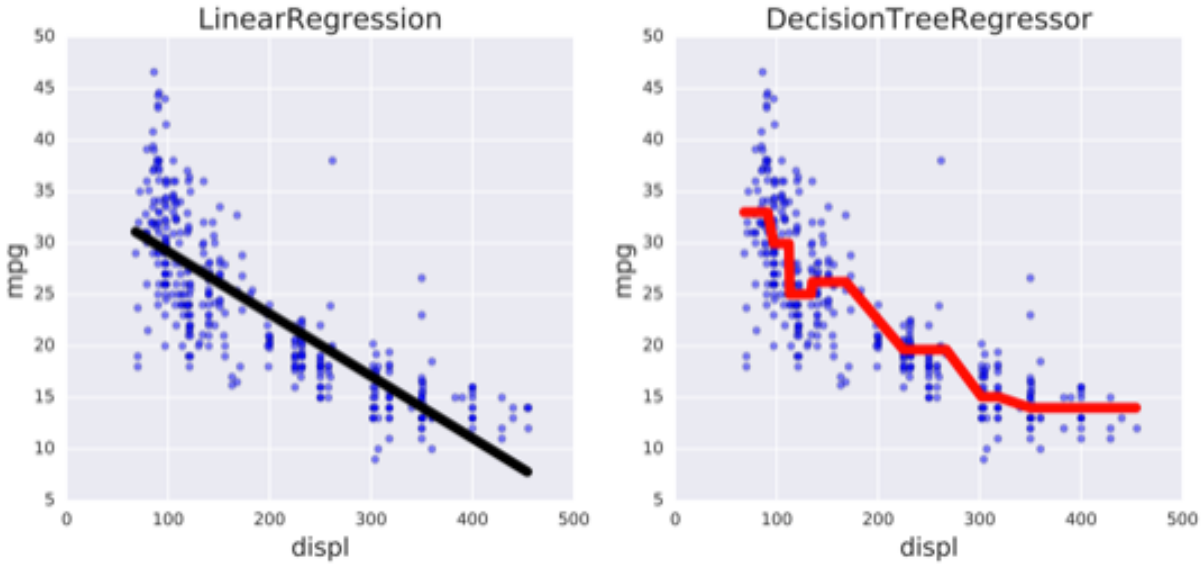
here mean squared error comparing actual vs mean, squaring residual to avoid negatives
**when a regression tree is trained on a dataset, the impurity of a node is measured using the mean-squared error of the targets in that node
this means that the regression tree tries to find the splits that produce leafs where in each leaf the target values are on average, the closest possible to the mean-value of the labels in that particular leaf

Prediction
as a new instance traverses the tree and reaches a certain leaf, its target-variable 'y' is computed as the average of the target-variables contained in that leaf

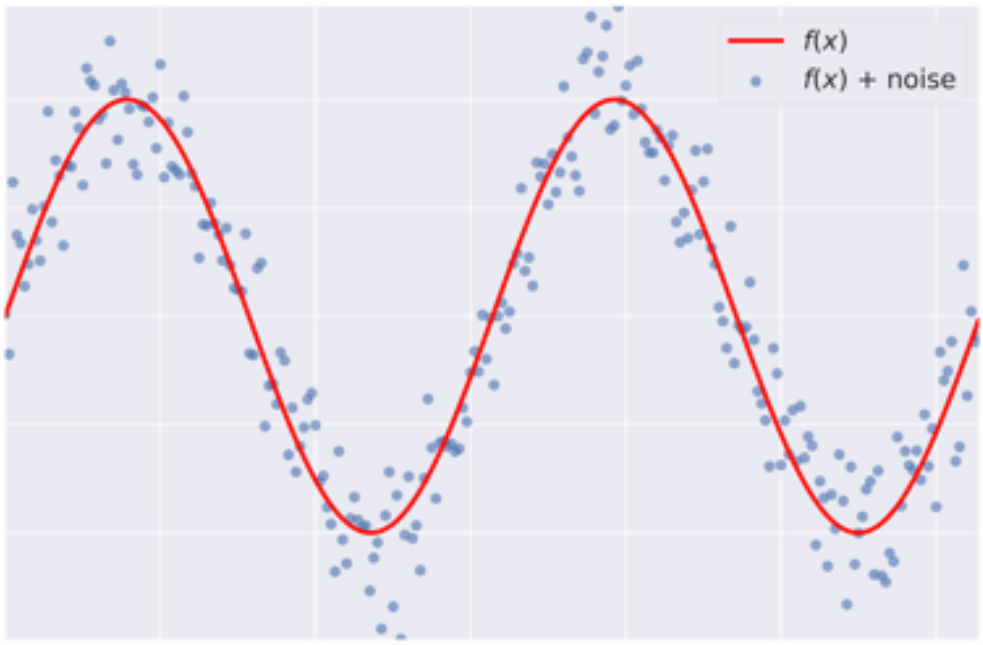
$$\hat{y}_{pred}(\text{leaf}) = \frac{1}{N_{leaf}} \sum_{i \in leaf} y^{(i)}$$

Reminder, "y hat" (ŷ) represents the predictions made by a machine learning model for the target variable, and it is used to differentiate these predicted values from the actual observed values (y) during the evaluation and analysis of the model's performance.

The power and flexibility of regression trees



Generalization Error of Supervised Learning
under the hood of supervised learning
supervised learning: f(x), f is unknown
y=f(x) represents the assumption that there's a mapping f between features and labels
f is an unknown function that you want to determine

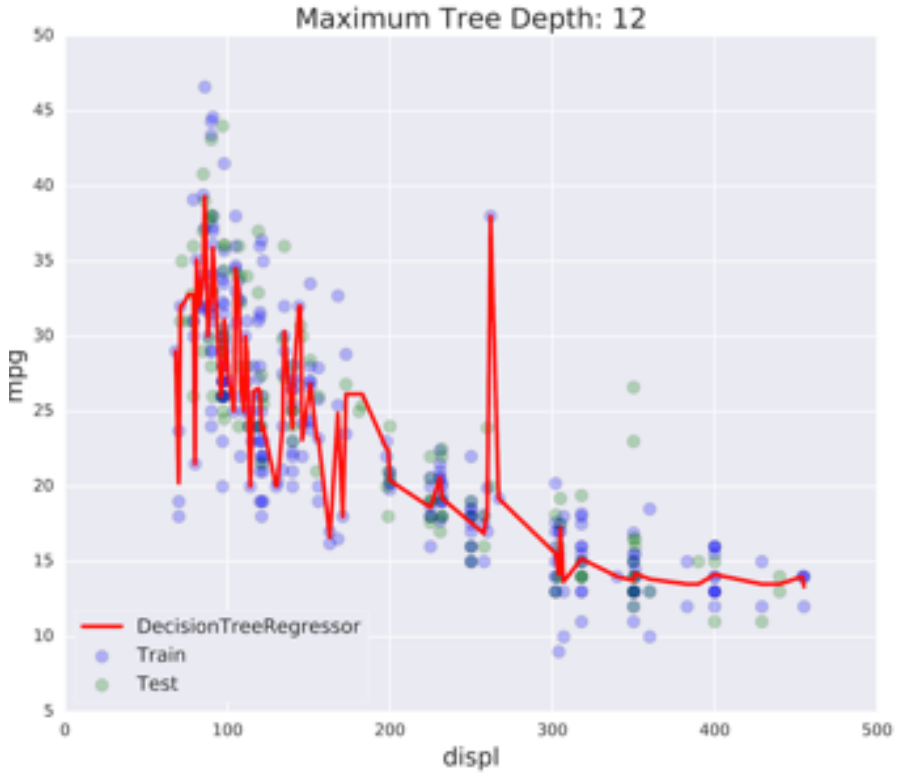


Goals of Supervised Learning
find a model f-hat that best approximates f
f-hat can be Logistic Regression, Decision Tree, Neural Network

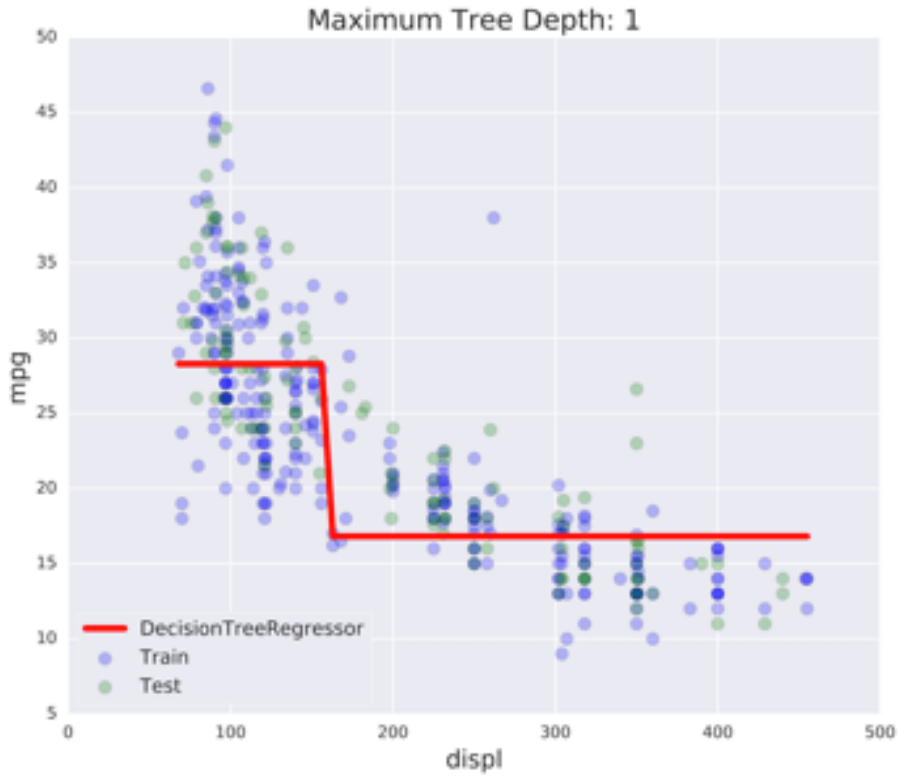
when training f-hat you want to make sure that noise is discarded as much as possible
end goal: f-hat should achieve a low predictive error on unseen datasets

Difficulties in approximating f
overfitting: f-hat(x) fits the training set noise
underfitting: f-hat is not flexible enough to approximate f

Overfitting
model memorizes the noise present in the training set
such models achieve a low training set error but often a high test set error

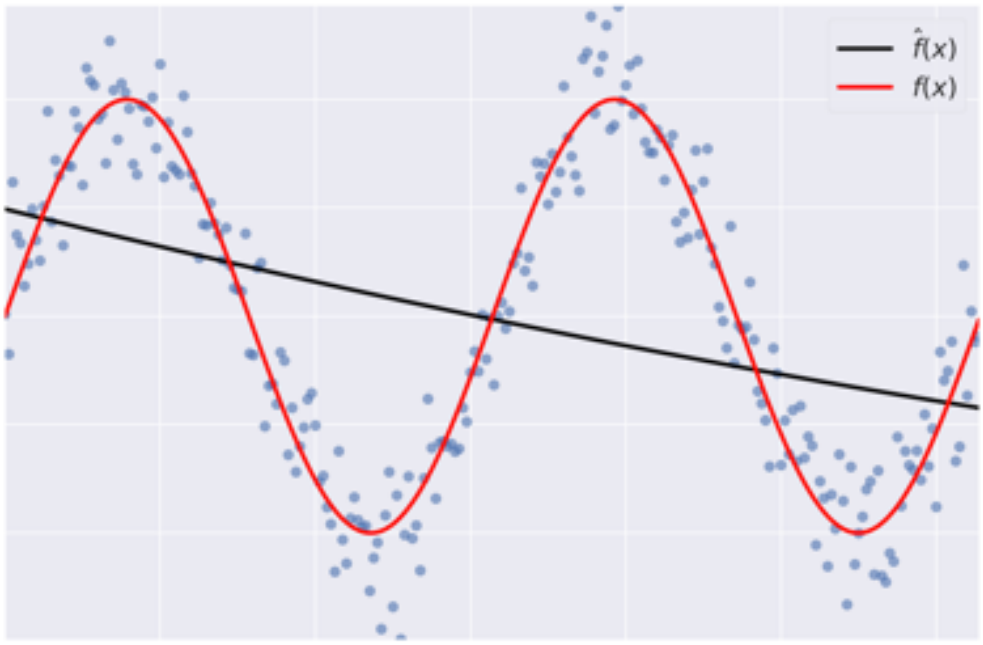


Underfitting
training set error is roughly equal to the test set error
however, both errors are relatively high
training mode is not flexible enough to capture teh complex dependency between features and labels

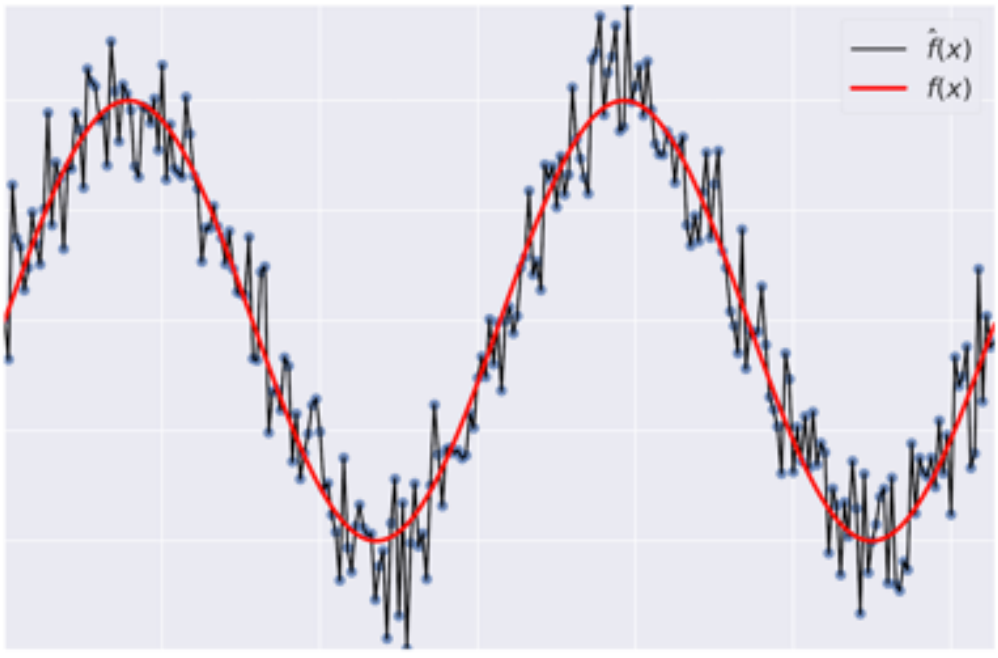


Generalization Error
generalization error of a model tells you how much it generalizes on unseen data
it asks, does f-hat generalize well on unseen data?
f-hat = bias^2 + variance + irreducible error
irreducible error is the error contribution of noise

Bias
bias error term tells you on average how much fhat and f are different
denoted: f-hat not equal sign f
high bias models lead to underfitting
below is an example of a high bias model:



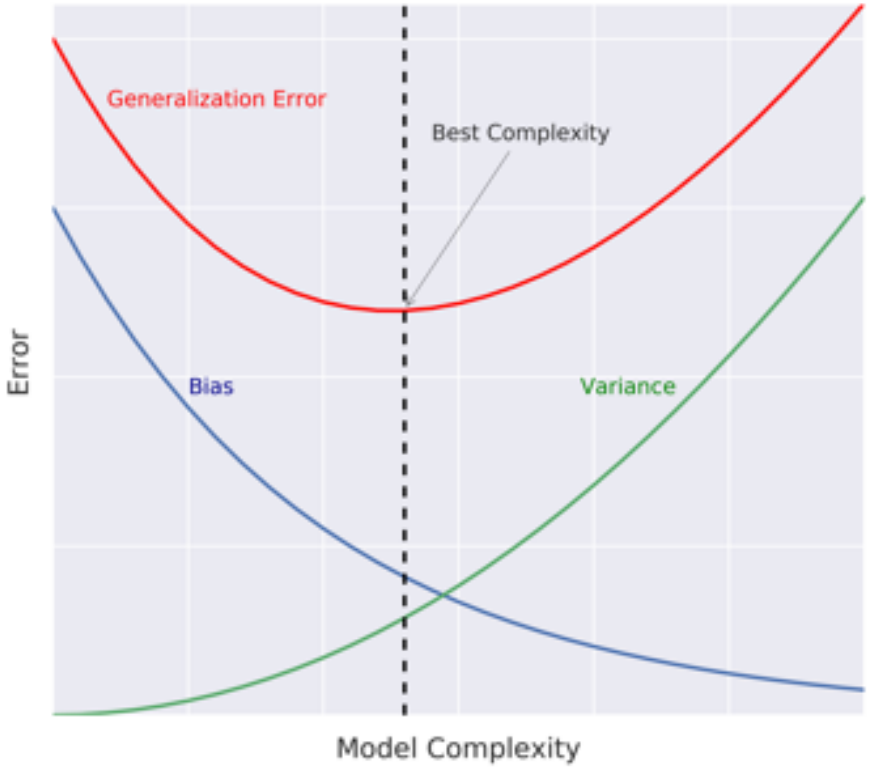
Variance
variance term tells you how much f-hat is inconsistent over different training sets
follows training points too closely
*this misses the true function
high variance models lead to overfitting
below shows a high variance model that misses the true function f shown in red



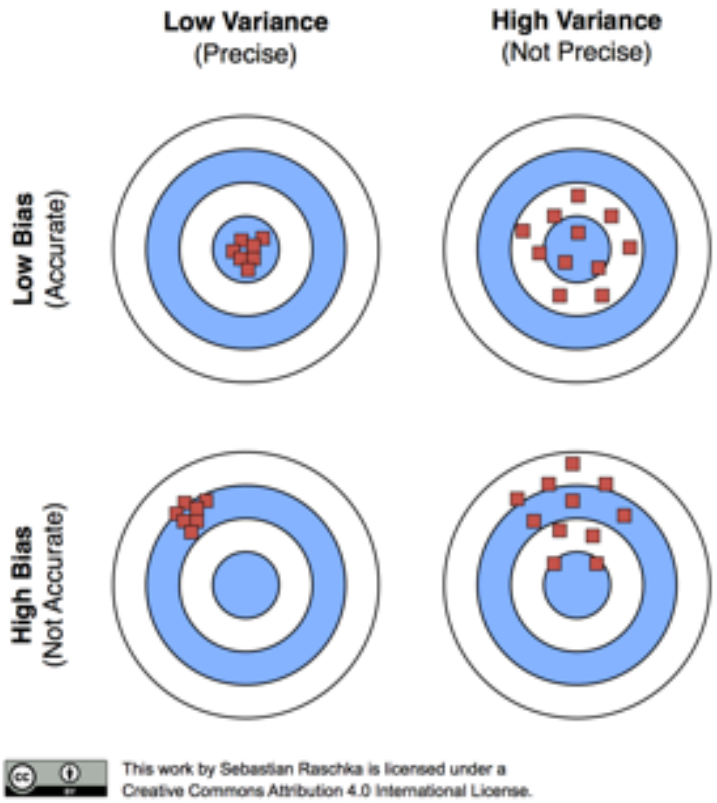
Model Complexity

complexity of a model sets its flexibility to approximate the true function f
examples of complexity > increasing max-tree depth, setting minimum samples per leaf, ...

*Bias-Variance Tradeoff
best model complexity corresponds to the lowest generalization error
when model complexity increases, the variance increases while the bias decreases
conversely, when the model complexity decreases, variance decreases and bias increases
generalization error is the sum of three terms with the irreducible error being constant
goal is to find a balance between bias and variance because as one increases the other decreases
this is what is known as the bias-variance trade-off



Bias-Variance Tradeoff: A Visual Explanation



How do you estimate the generalization error a model?
cannot be done directly because:

1. f is unknown
2. usually only have one dataset
3. noise is unpredictable so we don't have access to the error term

Solution

1. split the data to training and test sets
2. fit f-hat to the training set
3. evaluate the error of f-hat on the unseen test set

*generalization error of f-hat is rougly approximated by f=hat's error on the test set

Better model evaluation with cross-validation
test set should be kept untouched until one is confident about f-hat's performance
test set should only be used to evaluate f-hat's performance or error
*evaluating f-hat on training set will give a biased or optimistic estimation of the error because f-hat was already exposed to the training set when it was fit
to obtain a reliable estimate of f-hat's performance we use cross-validation
different types of CV > hold-out or K-fold

How does K-Fold CV work?
training set is split into partitions or ‘folds’
this number is set by the experimenter
k=10 will give 10 folds
the error of f-hat is evaluated 10 times on the 10 folds
each time, one fold is picked for evaluation after training f-hat on the other 9 folds
at the end, you’ll obtain a list of 10 errors

CV error = (E1+E2+…+_E10)/10
CV error is the mean of the 10 obtained errors

Variance Problems
Once you have computed f-hat’s cross-validation error, you can diagnose variance problems
if f-hat’s CV error is greater than f-hat’s training set error, f-hat is said to suffer from high variance
in this case f-hat has overfit the training set
this can be remedy by trying to decrease f-hat’s complexity
ways to do this
reduce the maximum-tree-depth or increase the maximum-samples-per-lear
can also gather more data to train fhat

Bias Problems
f-hat is said to suffer from high bias if its cross-validation-error is roughly equal to the training error but much greater than the desired error

in this case f-hat underfits the training set
ways to fix this
increase the model's complexity or gather more relevant features for the problem

Example - using K-Fold

```
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
SEED = 123
X_train, X_test, y_train, y_test = (X, y, test_size=0.3, random_state=SEED)
dt = DecisionTreeRegressor(max_depth=4, min_samples_leaf=0.14,
random_state=SEED)
#evaluate the list of MSE obtained by 10-fold CV
MSE_CV = - cross_val_score(dt, X_train, y_train, cv=10,
scoring='neg_mean_squared_error', n_jobs=-1)
#cross_val_score does not allow computing the mean-squared-errors directly
#n_jobs set to -1 exploits all available CPUs in computation (in this argument -1
represents all available CPU cores) can also set to 2, 3, 4 to use multiple CPUs in
parallel to speed up computation
#'neg_mean_squared_error' computes the negative-mean-squared-errors
output is a NumPy array of 10 negative MSEs achieved on the 10-folds
#multiply the result by -1 to obtain an array of CV-MSE
dt.fit(X_train, y_train)
#predict the labels of the training set
y_predict_train = dt.predict(X_train)
#predict the labels of the test set
y_predict_test = dt.predict(X_test)
#calculate CV MSE
print('CV MSE: {:.2f}'.format(MSE_CV.mean()))
#{:.2f} represents how to format a floating number > in this case with 2 decimals
and a colon ahead of the floating number
output > 20.51
#evaluate training set MSE
print('Train MSE: {:.2f}'.format
output > 15.30
#evaluate test set MsE
print('Test MSE: {:.2f}'.format(MSE(y_test, y_predict_test)))
output > 20.92
```

**given that the training set error is smaller than the CV-error we can deduce that dt overfits the training set and that it suffers from high variance
*also notice how CV MSE and Test MSE are roughly equal

Why is CV MSE and Test MSE roughly equal?
In a well-designed and representative cross-validation process, the Mean Squared Error (MSE) obtained during cross-validation and the MSE on the test set should be roughly equal. This similarity in values is expected under certain assumptions and indicates that the model's performance estimated during cross-validation is a good representation of its performance on unseen data.

The reasons why cross-validation MSE and test MSE are roughly equal are as follows:

1. Similar Data Distribution: Cross-validation involves splitting the dataset into multiple folds and training the model on different subsets of the data. The data distribution in each fold should be representative of the overall dataset. Similarly, the test set is also a representative sample of the entire dataset. Assuming the data distribution is consistent, the model's performance should be similar across both the cross-validation folds and the test set.

2. Limited Data Leakage: In a proper cross-validation setup, there should be limited data leakage between the training and validation folds. This means that the model is trained on one fold and tested on a different, non-overlapping fold, which helps prevent overfitting and ensures a more accurate estimation of the model's performance.

3. Sufficiently Large Dataset: If the dataset is sufficiently large, the cross-validation process provides a good estimation of the model's generalization performance. The more data points available, the more reliable the cross-validation estimates become.

4. Stable Model: If the model is stable and doesn't have a high variance, its performance should be consistent across different folds and the test set.

However, it's worth noting that slight variations can still exist between the cross-validation MSE and the test MSE due to the random splitting of the data during cross-validation and the limited size of the test set. These variations are generally acceptable, as long as they are not significant and the model's overall performance is consistent.

In summary, when cross-validation is conducted properly, the cross-validation MSE and test MSE should be approximately equal, which suggests that the model's performance estimates from cross-validation are reliable indicators of its performance on unseen data. This helps ensure that the model is not overfitting and can generalize well to new, unseen samples.

Why does cross_val_score not directly compute MSE?
cross_val_score from scikit-learn's **model_selection** module is a function primarily designed for computing and reporting the accuracy of a model when performing cross-validation. It is commonly used with classifiers where the evaluation metric is accuracy, which measures the proportion of correctly classified instances.
Mean Squared Error (MSE), on the other hand, is a metric typically used in regression tasks, where the goal is to predict continuous values. MSE measures the average squared difference between the predicted values and the actual target values. Unlike accuracy, which is a classification metric, MSE is a regression metric.
Since **cross_val_score** is designed for classification tasks and focuses on accuracy, it does not directly compute MSE. Instead, it calculates and reports the accuracy score based on the cross-validated predictions. However, scikit-learn provides other functions like **cross_val_predict** and **cross_val_score** with **scoring='neg_mean_squared_error'** that can be used for computing the negative mean squared error (neg-MSE) for regression tasks. The negative MSE is used because scikit-learn optimizes for higher scores, and negating the MSE makes it consistent with that behavior.

Why scikit-learn uses negation for certain scoring metrics?
In scikit-learn, many scoring metrics, including Mean Squared Error (MSE), are designed to be maximized when the model's performance is better. However, some algorithms in scikit-learn, particularly those used for optimization or cross-validation, are designed to work in a way where they try to minimize the objective function. For example, when using cross-validation or grid search to find the best hyperparameters for a model, the process aims to find the hyperparameters that minimize the scoring metric.

Since many performance metrics are designed to be maximized but scikit-learn's optimization algorithms work to minimize the objective, a simple workaround is to take the negative of the scoring metric. By doing so, the optimization process can now work to minimize the negative value, which is equivalent to maximizing the original value.

Here's a step-by-step explanation of why negation is used for certain scoring metrics like MSE:

1. Original objective: For metrics like MSE, lower values are better, as they indicate smaller prediction errors and better model performance. We want to minimize MSE during the optimization process.

2. Optimizing for higher values: Scikit-learn's optimization algorithms are designed

to find the best hyperparameters by minimizing the objective function.

3. Negation: To work around the fact that scikit-learn optimizes for higher values, we can simply negate the original metric. Now, the optimization algorithms will work to minimize the negative of the original metric, effectively maximizing the original metric.

For example, if the original MSE is 10, the negative MSE will be -10. When the optimization algorithm minimizes -10, it is, in fact, maximizing the original MSE of 10, which is the desired behavior.

To summarize, using negation for certain scoring metrics in scikit-learn is a convenient way to align the optimization process, which aims to minimize the objective function, with the goal of maximizing the performance of the model as measured by those metrics.

Advantages of classification and regression trees (CARTs)

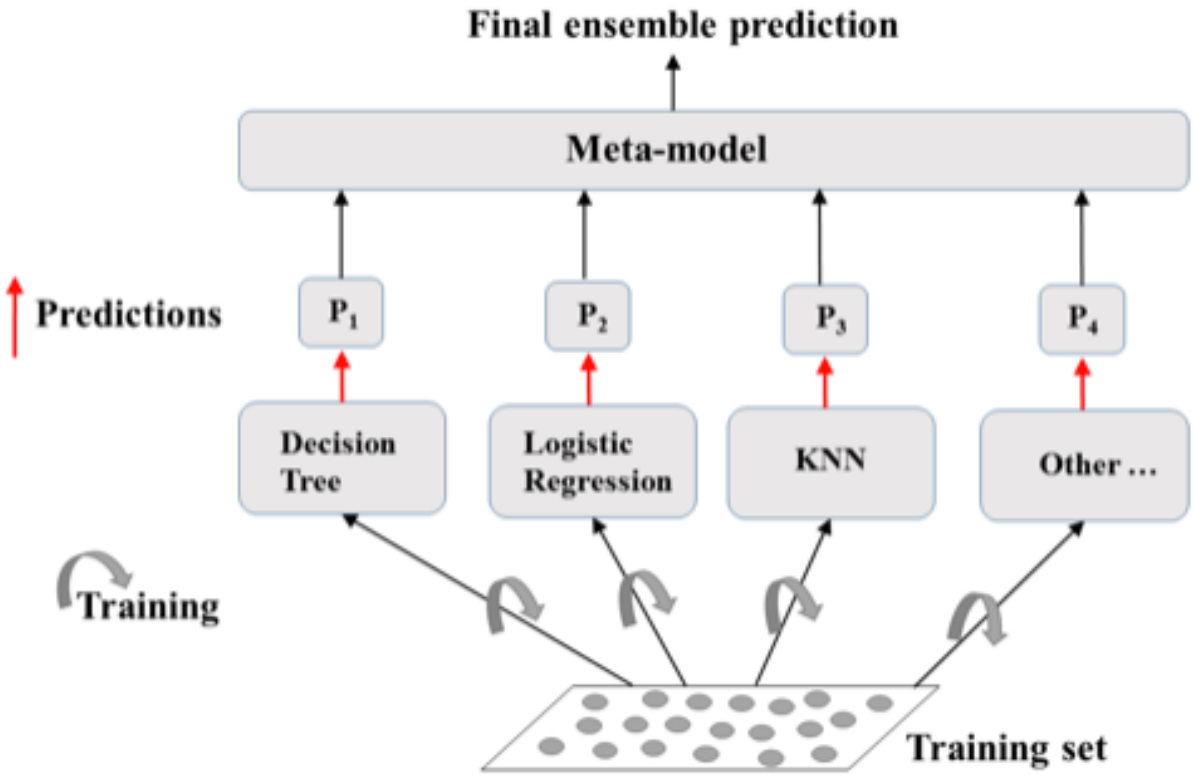
- simple to understand
- output is simple to interpret
- easy to use
- their flexibility give them an ability to describe nonlinear dependencies between features and labels
- do not need extensive feature preprocessing to train a CART (ie. no need to standardize or normalize features before feeding them to a CART

Limitations of CARTs

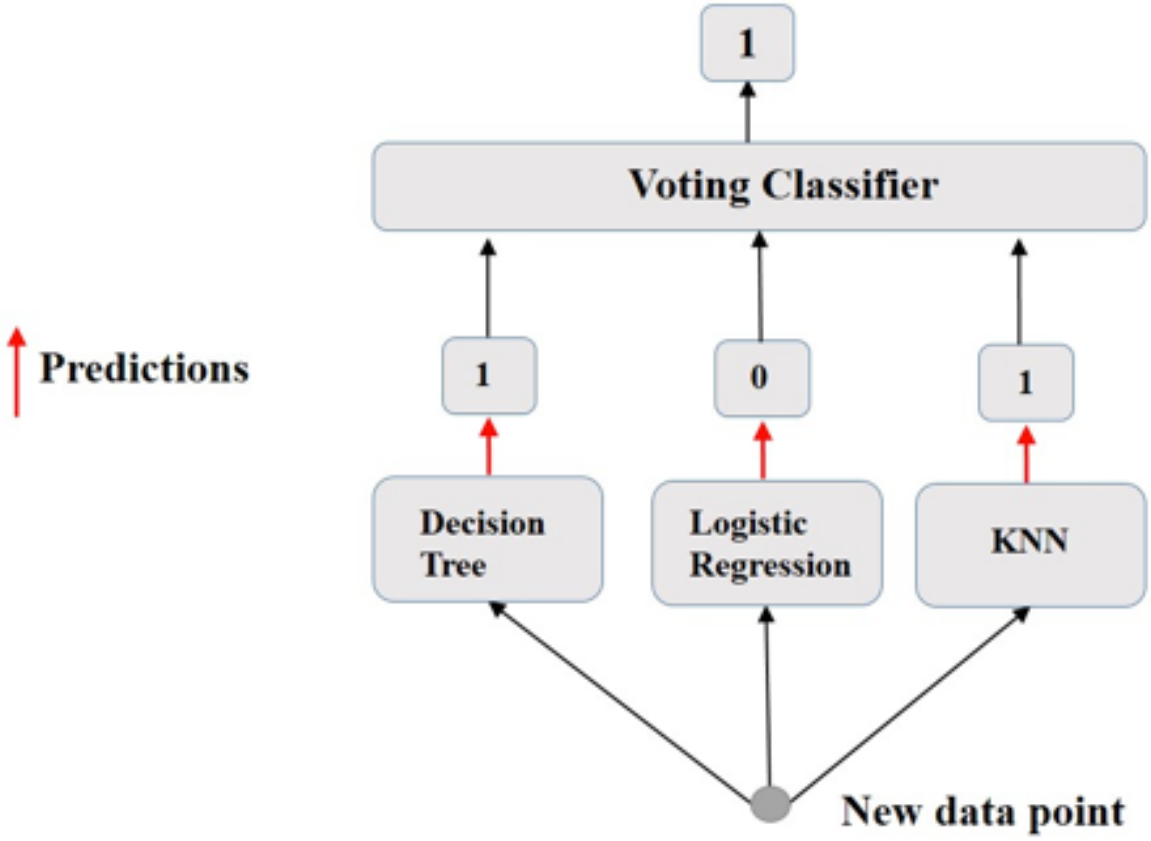
- a classification tree is only able to produce orthogonal decision boundaries
- sensitive to small variation in the training set (ie a single point removed from the training set can drastically change a CART's learned parameters
- CARTs also suffer from high variance when they are trained without constraints (may cause overfitting)

Ensemble Learning
this is a solution that takes advantage of the flixibility of CARTs while reducing their tendency to memorize noise

- different models are trained on the same dataset
- each model makes its own predictions
- a meta-model then aggregates the predictions of individual models
- then outputs a final prediction which is more robust and less prone to errors that each individual model
- best results occur when the models are skillful but in different ways



Ensemble technique: Voting Classifier
take a binary classifciation task
the ensemble here consists of N classifiers making the predictions P0, P1, P2,..., PN with Pi = 0 or 1
*meta-model outputs the final prediction by hard voting
visual example of hard voting:

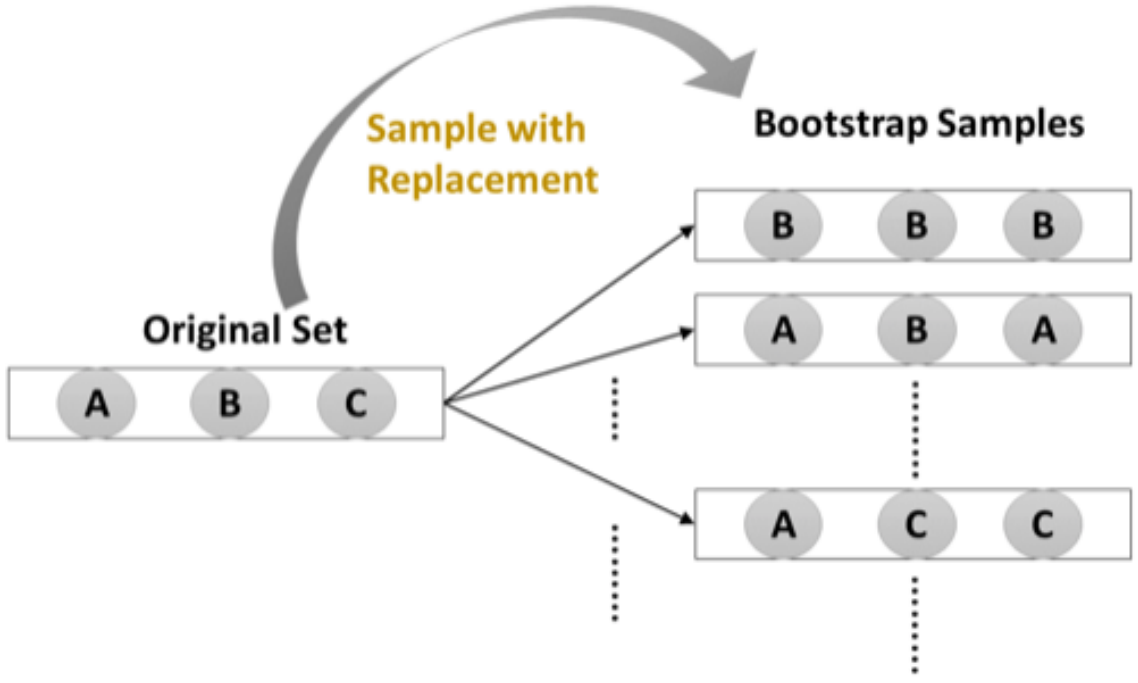


3 classifiers, 2 picked 1, 1 picked 0, so 2 votes to 1 vote, voting classifier predicts 1

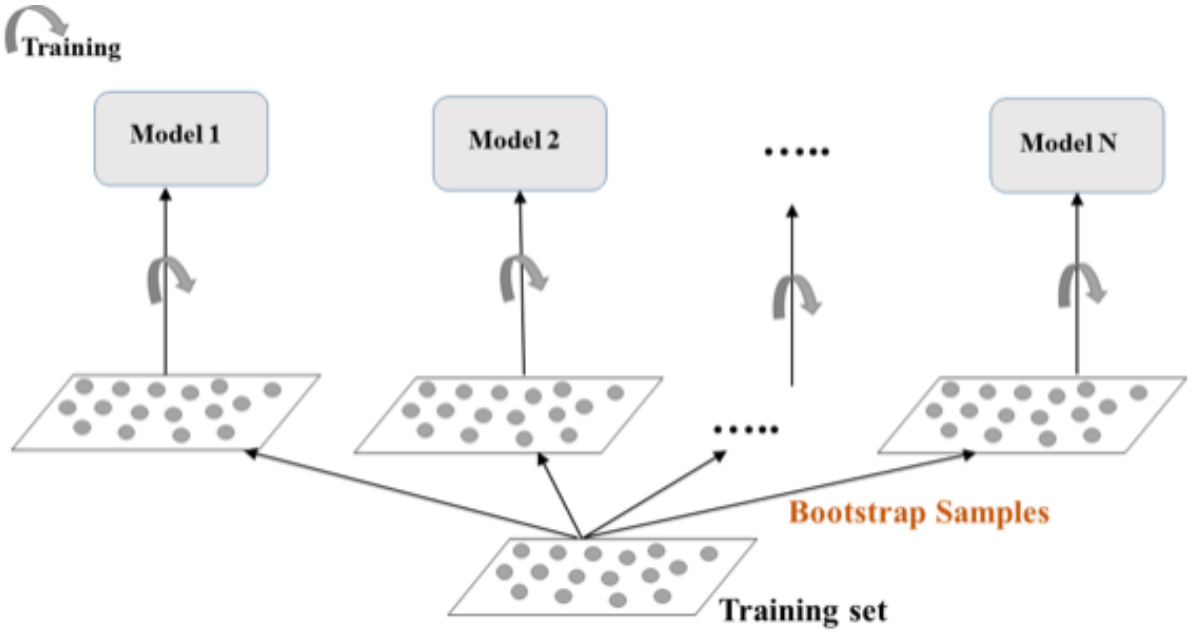
```
from sklearn.ensemble import VotingClassifier
SEED = 1
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=SEED)
#instantiate individual classifiers
lr = LogisticRegression(random_state=SEED)
knn = KNN()
dt = DecisionTreeClassifier(random_state=SEED)
#define a list called classifier that contains the tuples (classifier_name, classifier)
classifiers = [('Logisitic Regression, lr), ('K Nearest Neighbors', knn),
('Classification Tree', dt)]
#use a for loop to iterate over the list 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)))
```

output > showed lr with the highest accuracy of approx 94%
#now intsantiate a VotingClassifier ‘vc’
vc = VotingClassifier(estimators=classifiers)
#fit ‘vc’ to the training 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)))
output > approx 95% which is better than all the individual classifier accuracy scores

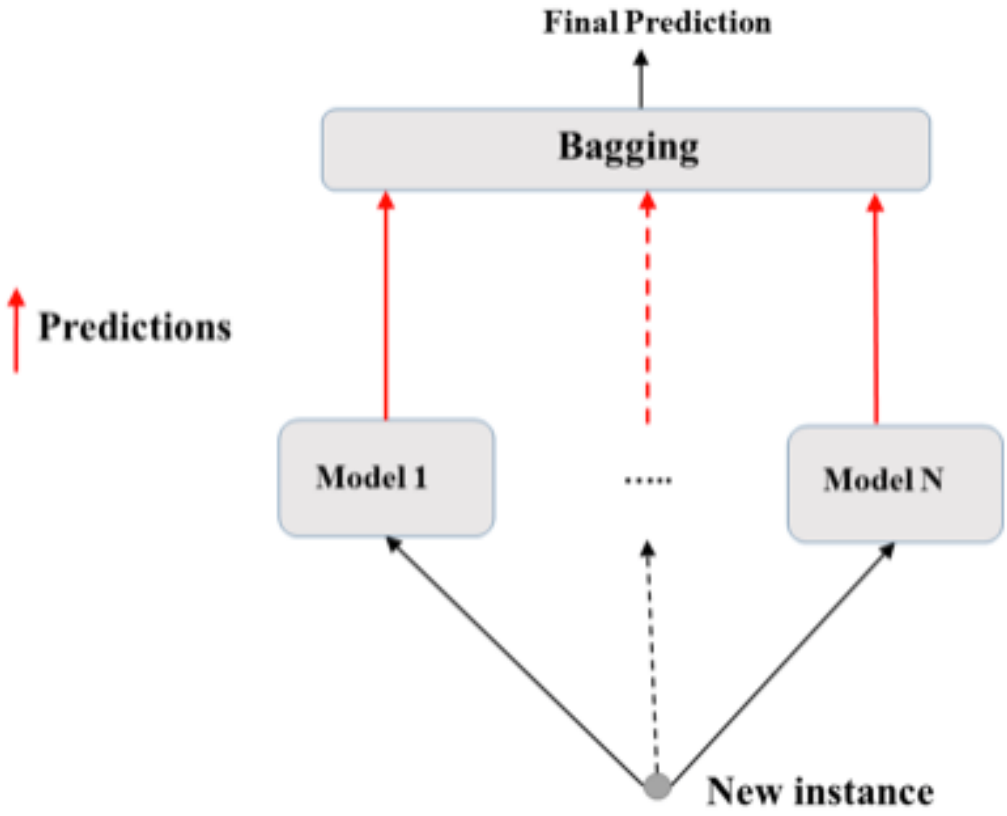
Bagging
an ensemble method
also known a bootstrap aggregation
why? > uses a technique known as the bootstrap
formed by models that use the same training algorithm
however these models are not trained on the entire training set
each model is trained on a different subset of the data
bagging has the effect of reducing the variance of individual models in the ensemble



another nice visual explaining bagging:



Bagging prediction:



the meta model collects these predictions and outputs a final prediction depending on the nature of the problem.

Bagging in classification
final prediciton aggregates predictions by majority voting
class_bag = BaggingClassifier()

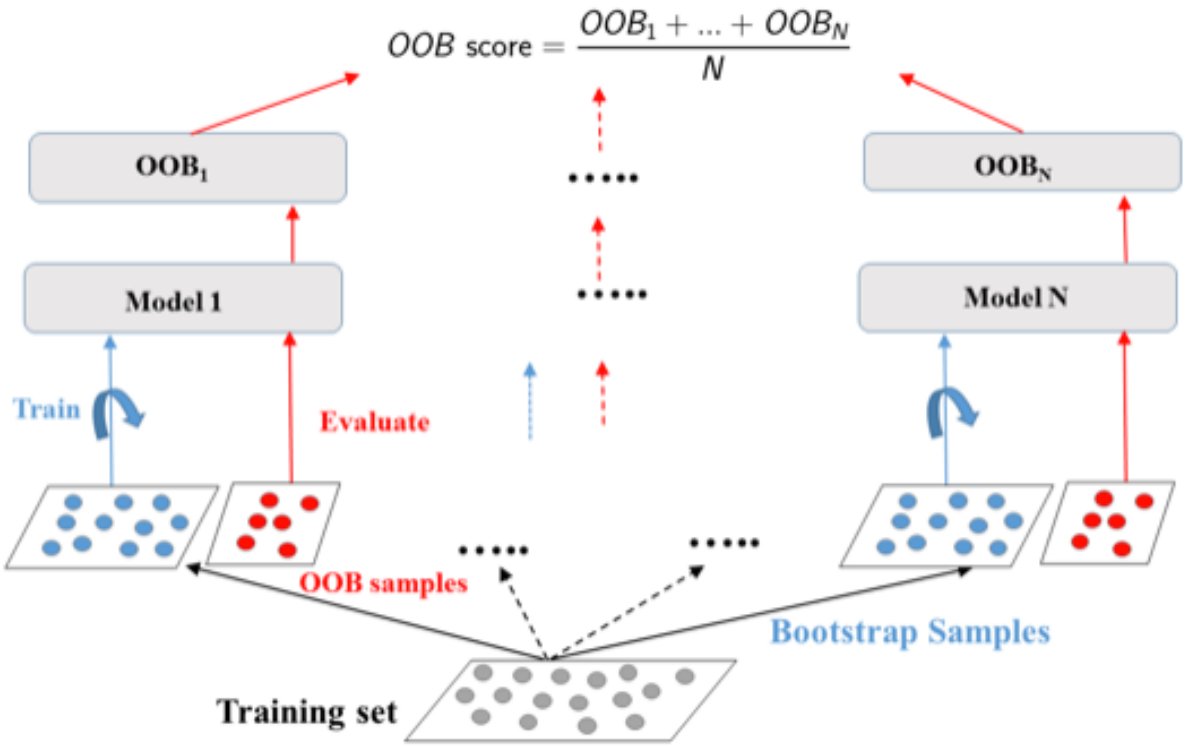
Bagging in regression
final prediction is the average of the predictions made by the individual models forming the ensemble
reg_bag = BaggingRegressor()

example - Breast Cancer dataset
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
SEED = 1
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, stratify=y, random_state=SEED)
#reminder on 'stratify' argument, used when you have imbalanced class distributions it the target variable (y)
#when set to 'y', it ensures that the class distribution in both the training and testing datasets remain similar to the original class distribution in the full dataset
#you are telling train_test_split to split the dat in a way that preserves the proportions of the different classes in both the training and testing sets
#instantiate DecisionTreeClassifier
dt = DecisionTree Classifier(max_depth=4, min_samples_leaf=0.16, random_state=SEED)
#instantiate BaggingClassifier that consists of 300 samples of the above DecisionTreeClassifier (dt)
#'base_estimator' argument defines the dataset, n_estimators defines the desired amount of samples
#remember all samples are with replacement
#again n_jobs set to -1, where -1 stands for all (uses all CPU cores)
bc = BaggingClassifier(base_estimator=dt, n_estimators=300, n_jobs=-1)
bc.fit(X_train, y_train)
y_pred = bc.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy of Bagging Classifier: {:.3f}'.format(accuracy))

Out of Bag Evaluation
bagging - some instances may be sampled several times and others not at all for one model
on average for each model 63% of the training instances are sampled
the remaining 37% that are not sampled constitute what is known as Out-of-bag

(or OOB) instances
*since OOB instances are not seen by a model during training, these can be used to estimate the performance of the ensemble without the need for cross-validation this technique is known as OOB-evaluation

OOB Evaluation



how to add OOB score with sklearn
**follow example as above
continuing with same sample - breast cancer dataset classifying malignant and benign tumors
bc = Bagging Classifer(base_estimator=dt, n_estimators=300, oob_score=True, n_jobs=-1)
#oob_score set to True evaluates the OOB accuracy of bc after training
**OOB-score corresponds to the accuracy for classifiers and the r-squared score for regressors
#extract OOB accuracy
oob_accuracy = bc.oob_score_
print('OOB accuracy: {:.3f}'.format(obb_accuracy))

More on OOB:
The term "oob_score" is related to the Out-of-Bag (OOB) error estimation in ensemble learning, specifically in the context of random forests.

In random forests, each decision tree in the ensemble is trained on a bootstrapped

sample of the original data, meaning that some data points are not included in the training set for each tree. The data points that are not used for training a particular tree are referred to as "out-of-bag" (OOB) samples.

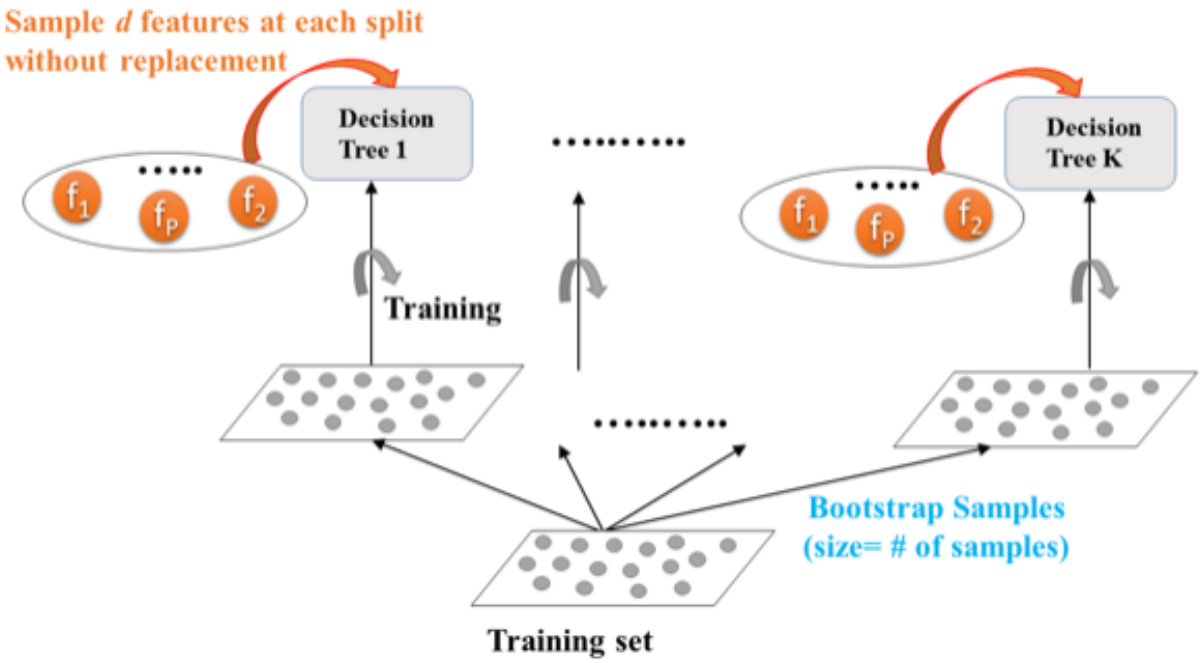
The OOB error is an estimate of the model's generalization performance, calculated using the OOB samples. For each data point in the dataset, the OOB error is computed by predicting its target value using only the trees that were not trained on that specific data point.

The "oob_score" is a parameter in scikit-learn's RandomForestClassifier and RandomForestRegressor classes that allows you to enable or disable the calculation of the OOB error during the training of a random forest model. When "oob_score=True," scikit-learn will automatically compute the OOB error as the model is trained.

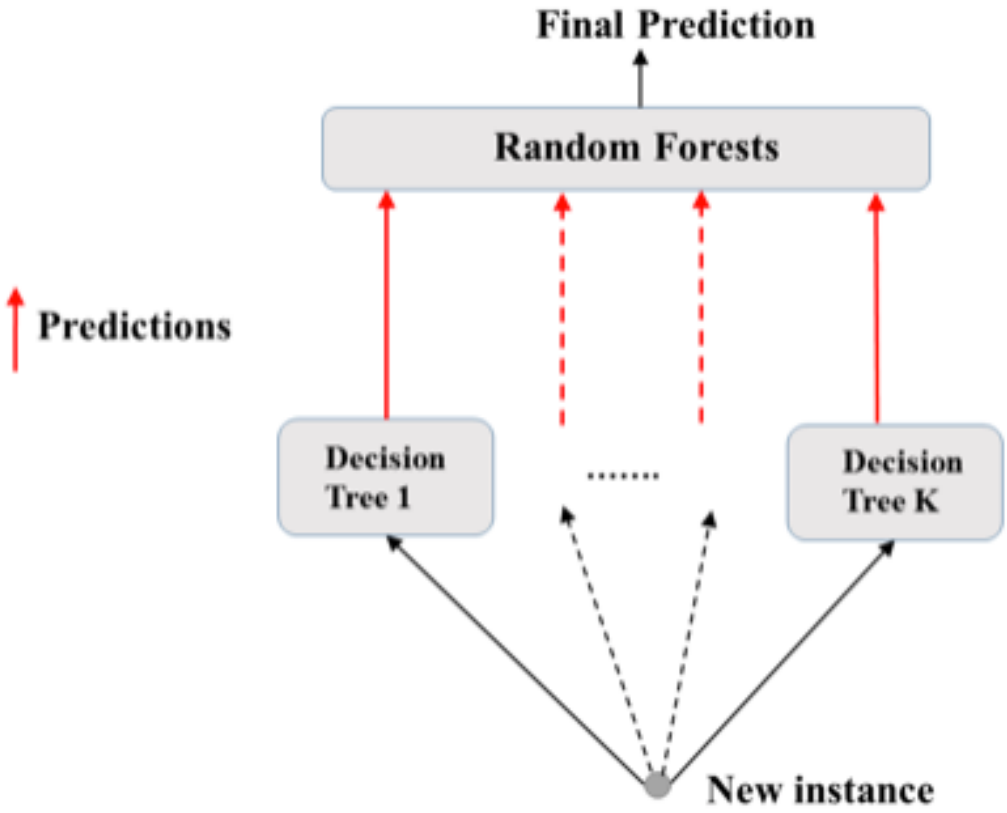
The OOB error serves as an estimate of the model's performance on unseen data and can be helpful for assessing the model's generalization ability without the need for an explicit train-test split.

The OOB error can be a useful alternative to traditional cross-validation for model evaluation when using random forests, especially when dealing with large datasets. However, keep in mind that it is not always a perfect substitute for more comprehensive evaluation methods, and traditional cross-validation should still be used in certain cases for a more thorough assessment of the model's performance.

Random Forests
another ensemble learning method
recall that in bagging the base estimator can be any model including a decision tree, logistice regression, neural net,...
random forests uses a decision tree as a base estimator
each estimator is trained on a different bootstrap sample having the same size as the training set
random forest introduces further randomization that baggin when training each of the base estimators
when each tree is trained, only d features can be sampled at each node without replacement
where d is < total number of features



each tree forming the ensemble is trained on a different bootstrap sample from the training set
when a tree is trained, at each node, only d features are sampled from all features without replacement
node is then split using the sampled feature that maximizes information gain
in scikit, d defaults to the square root number of features (ex 100 features > 10 are sampled at each node)



meta-classifer collects predictions and makes a final prediction
classification > majority voting
RandomForestClassifer()
regression > average of all the labels predicted by the base estimators
RandomForestRegressor()
*in general, random forests achieves a lower variance than individual tres

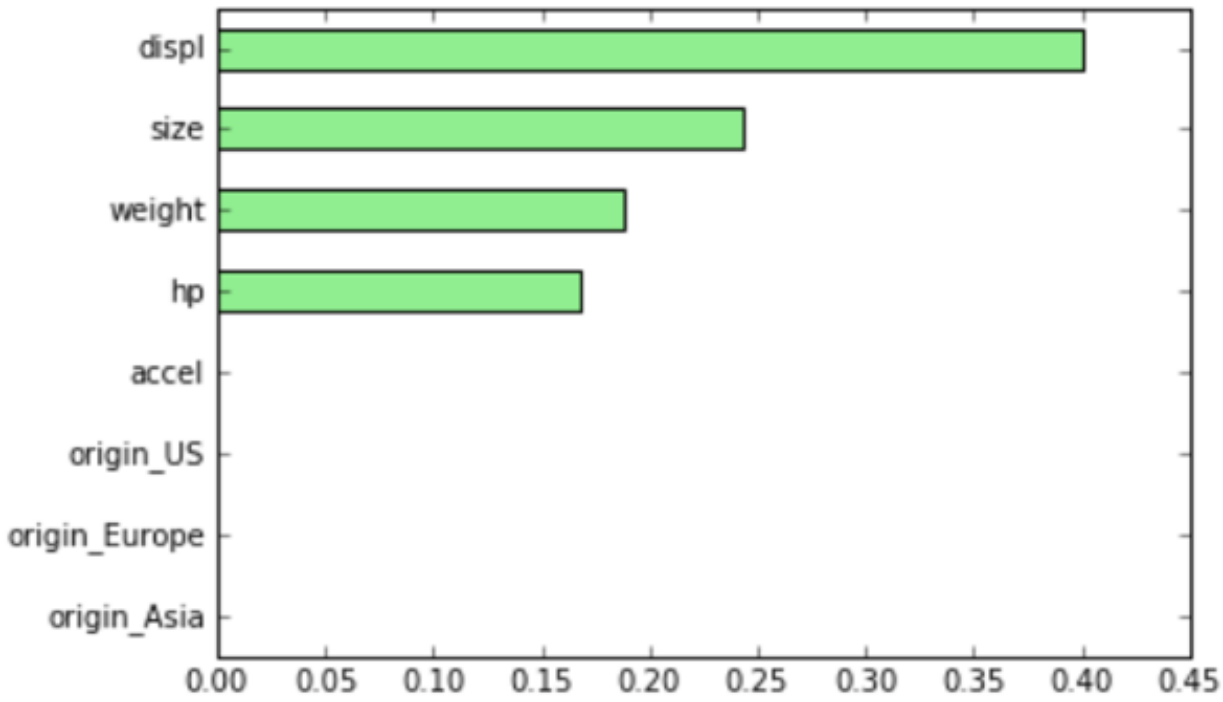
example

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error as MSE
SEED = 1
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=SEED)
rf = RandomForestClassifier(n_estimators=400, min_samples_leaf=0.12,
random_state=SEED)
rf.fit(X_train, y_train)
#predict the test set labels
y_pred = rf.predict(X_test)
#evaluate the RMSE for test set
rmse_test = MSE(y_test, y_pred)**(1/2)
print('Test set RMSE of rf: {:.2f}'.format(rmse_test))
```

Feature Importance
when a tree based method is trained, the predictive power of a feature or its importance can be assessed
in scikit, feature importance is assessed by measuring how much the tree nodes use a particular feature to reduce impurity
importance of a feature is expressed as a percentage indicating the weight of that feature in training and prediction (weighted average)
once model is trained in scikit, features importances can be assessed by extracting the feature_importance_ attribute
visualize:

```
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)
sorted_importances_rf = importances_rf.sort_values()
#make a horizontal bar plot
sorted_importances_rf.plot(kind='barh', color='lightgreen')
plt.show()
```

output >

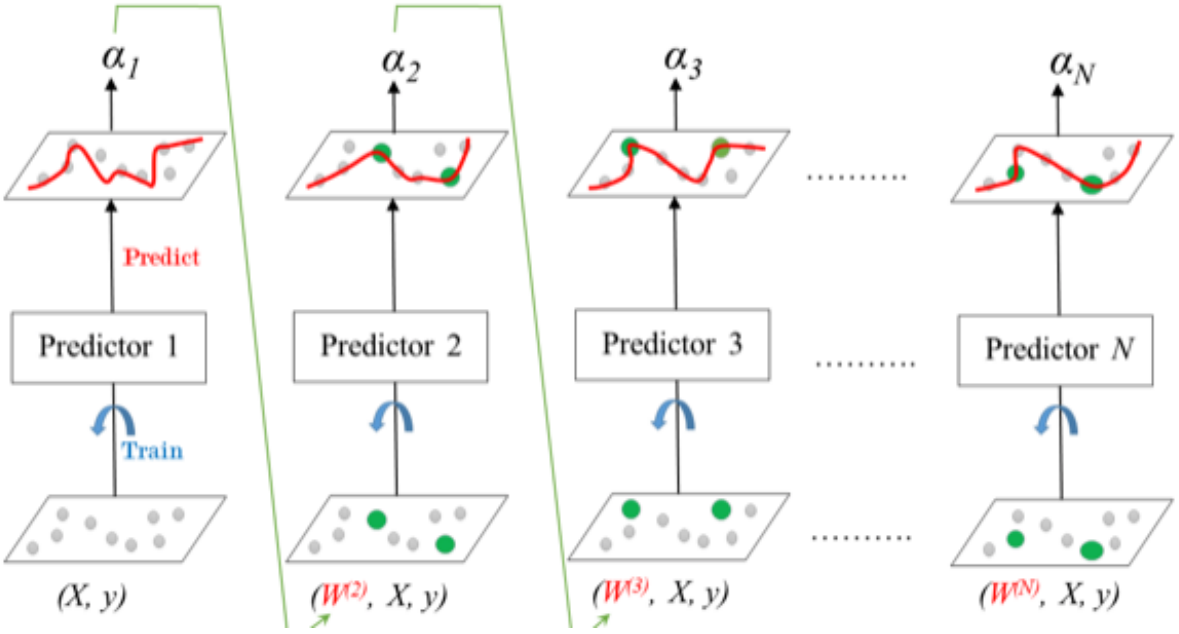


shows that according to our model, features displ, size, weight, and hp are the most predictive features

Boosting
another ensemble method

which many predictors are trained and each predictor learns from the errors of its predecessor
'boosting' refers to the strength or improvement in numbers
the concept of combining several weak learners to form a strong learner
what is a weak learner
a model doing just slightly better than random guessing
example a decision tree with only a max_depth of 1 (also referred to as a decision-stump)
in boosting, an ensemble of predictors are trained sequentially and each predictor tries to correct the errors made by its predecessor
mulitple bootsing methods (popular ones are AdaBoost and Gradient Boosting)

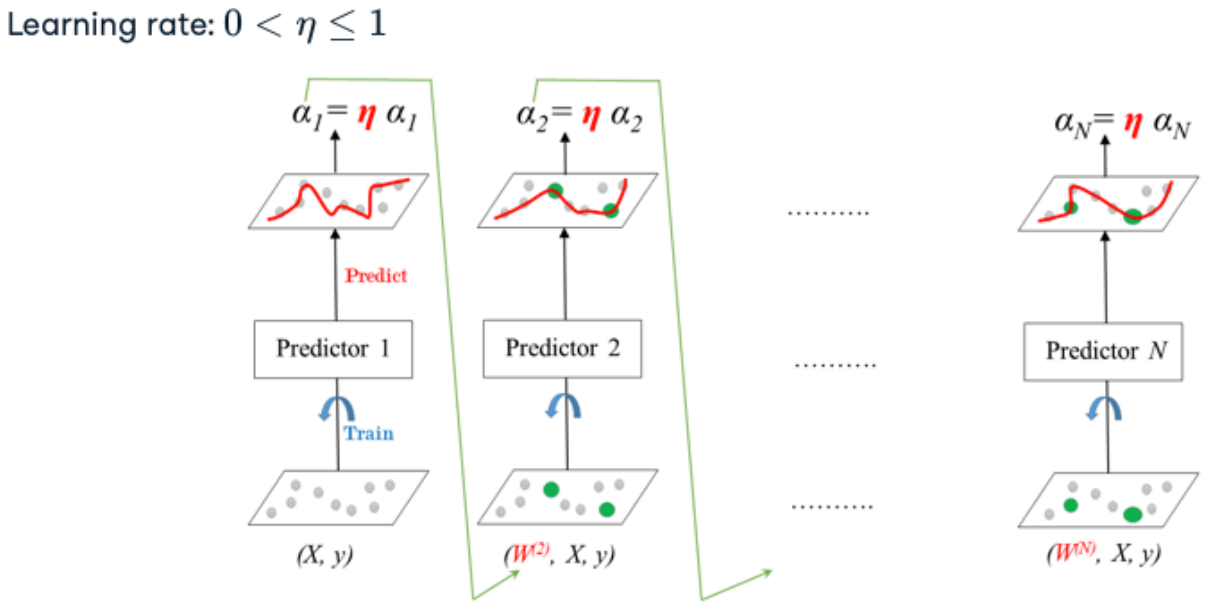
AdaBoost
stands for adaptive boosting
each predictor pays more attention to the instances wrongly predicted by its predecessor
does this by constantly changing the weights of training instances
each predictor is assigned a coefficient alpha
alpha weighs its contribution in the ensemble's final prediction
alpha depends on the predictor's training error
**be aware, appears that the name 'alpha' as the designation for AdaBoost's training error is not universally adopted
great visual to explain:



Predictor1's training error is alpha1 (Predictor1's coefficient) which is then used to determine the weights on the instances for Predictor2
the weights of the training instances can be seen as green on the above visual
during training the model is forced to pay more attention to these green areas

(weighted instances)

Learning Rate
also called eta
range is 0 to 1
used to shrink the coefficient alpha of a trained predictor
*a trade-off between eta and the number of estimators
smaller value of eta should be compensated by a greater number of estimators
higher learning rate at higher chance for overfitting
determines ‘step size’, ie how fast to learn
larger step size tells the model to take each model design more literally
visualize where eta is in the process:



more on learning rate and AdaBoost training error (sometimes referred to as alpha):
In the context of machine learning, "eta" refers to the learning rate parameter in gradient boosting algorithms, such as Gradient Boosting Machines (GBM) and XGBoost.

The learning rate, denoted as "eta" or "learning_rate," is a hyperparameter that controls the step size at each iteration when building the ensemble of weak learners (usually decision trees). It determines how much the model adjusts its predictions based on the errors of the previous weak learner.

When using gradient boosting, each iteration (boosting round) builds a new weak learner to correct the errors made by the previous ensemble. The learning rate plays a crucial role in this process. A smaller learning rate slows down the learning

process, as each tree in the ensemble makes smaller adjustments to the predictions. On the other hand, a larger learning rate speeds up learning, as each tree has a larger influence on the final prediction.

Now, let's address the second part of your question, which involves alpha. It seems there might be some confusion here. In gradient boosting, "alpha" is not directly related to the learning rate; it typically refers to the L1 regularization term in the context of regularized linear models, such as Lasso regression.

To clarify further:

1. Learning Rate (eta): As explained above, the learning rate controls the step size during gradient boosting iterations. A smaller learning rate makes the model more robust to overfitting but requires more iterations to converge. A larger learning rate may lead to faster convergence but increases the risk of overfitting.

2. Alpha (L1 Regularization): In the context of L1-regularized linear models like Lasso regression, "alpha" is the regularization strength parameter. It introduces a penalty term to the loss function, encouraging the model to prefer simpler, more sparse solutions. Higher values of alpha lead to more aggressive feature selection, resulting in more coefficients being exactly zero.

In summary, "eta" or the learning rate is used in gradient boosting algorithms, controlling the step size at each iteration. On the other hand, "alpha" typically refers to the L1 regularization strength in regularized linear models like Lasso regression, promoting sparsity in the model's coefficients. The two parameters are different and used in different contexts.

AdaBoost: Prediction
similar to some of the above ensemble methods
Classification > weighted majority voting
AdaBoostClassifier()
Regression > a weighted average

**note individual predictors do not need to be CARTs
however, CARTs are often used because of their high variance

example - Breast CA dataset evaluation with ROC-AUC score
from sklearn.ensemble import AdaBoostClassifer
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import train_test_split
SEED = 1

```
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 = DecisionTreeClassifier(max_depth=1, random_state=SEED)
#instantiate an AdaBoost classifier consisting of 100 decision stumps
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 probability of obtaining the positive class in the test set
#need this component in order to evaluate the ROC-AUC score
y_pre_proba = adb_clf.predict_proba(X_test)[:,1]
adb_cllf_roc_auc_score = roc_auc_score(y_test, y_pred_proba)
```

More on ROC-AUC:
The Receiver Operating Characteristic Area Under the Curve (ROC-AUC) score is a performance metric used to evaluate the quality of binary classification models. It assesses the model's ability to discriminate between positive and negative samples by measuring the area under the ROC curve.

To calculate the ROC-AUC score, you need the probability of the positive class predictions rather than just the predicted class labels. This is because the ROC curve is created by plotting the True Positive Rate (TPR) against the False Positive Rate (FPR) at various probability thresholds for classifying positive instances.

Here's a step-by-step explanation of why probabilities are needed:

1. Probability Thresholding: When you have the probabilities of the positive class, you can set different probability thresholds to determine which instances are classified as positive and which as negative. For example, if the threshold is set at 0.5, any instance with a probability of the positive class greater than or equal to 0.5 will be predicted as positive, and the rest as negative.

2. TPR and FPR Calculation: With the probability thresholding in place, you can calculate the True Positive Rate (TPR) and False Positive Rate (FPR) at each threshold. TPR measures the proportion of positive instances that are correctly classified as positive (i.e., true positives), while FPR measures the proportion of negative instances that are incorrectly classified as positive (i.e., false positives).

3. ROC Curve: The ROC curve is created by plotting the TPR against the FPR for various probability thresholds. Each point on the ROC curve represents the trade-off between the true positive rate and the false positive rate at a specific threshold.

4. ROC-AUC Score: The ROC-AUC score is then calculated as the area under the ROC curve. It provides a single scalar value that quantifies the model's overall discriminatory power. A perfect model has an AUC score of 1, while a random model has an AUC score of 0.5.

By considering the probabilities of the positive class, the ROC-AUC score takes into account the model's ability to rank positive instances higher than negative instances, making it a more comprehensive evaluation metric for binary classification models than accuracy alone.

In Python, when evaluating the ROC-AUC score, you typically provide the probability estimates of the positive class (not the predicted labels) to the `roc_auc_score` function in scikit-learn or other libraries. This ensures that the evaluation is based on the model's confidence in its predictions rather than just binary labels.

More on positive classes in conjunction with ROC-AUC:
No, the positive class is not always in the second column. The arrangement of classes in the columns depends on how the classes are encoded or represented in the specific context or dataset.

In binary classification, where there are only two classes (positive and negative), there are two common ways to encode the classes:

1. Label Encoding: In label encoding, one of the classes is represented as 0 and the other as 1. The positive class can be either 0 or 1, and the negative class will have the opposite label. For example, if the positive class is represented as 1, the negative class will be represented as 0, and vice versa.

2. One-Hot Encoding: In one-hot encoding, each class is represented as a binary vector with one "1" and all other elements as "0." The position of the "1" in the vector determines the class. For binary classification, one of the classes is represented as [1, 0], and the other as [0, 1]. The positive class can be either [1, 0] or [0, 1], and the negative class will have the opposite encoding.

The choice of encoding (label encoding or one-hot encoding) is typically determined by the machine learning framework or library being used and the specific problem being addressed.

For example, in scikit-learn, binary classification tasks often use label encoding, where the positive class is represented as 1, and the negative class is represented as 0. In contrast, in other libraries like TensorFlow or Keras, one-hot encoding is commonly used for binary classification tasks, with the positive class encoded as

[1, 0] and the negative class encoded as [0, 1].

It's important to note that regardless of the encoding used, the ROC-AUC score or other performance metrics can handle either encoding scheme and provide accurate evaluations of the model's performance. The key is to ensure consistency in the encoding throughout the data preprocessing and evaluation process.

More on .predict_proba():
The `.predict_proba()` method is a function provided by many machine learning models in scikit-learn and other libraries. It is used to obtain the predicted probabilities for each class label in a classification task.

In binary classification, a model predicts one of two possible class labels (e.g., positive and negative). The `.predict_proba()` method returns the probabilities that an instance belongs to each class label. The output is an array of shape `(n_samples, n_classes)`, where `n_samples` is the number of data points and `n_classes` is the number of classes.

For example, let's say you have a binary classification model and you want to predict the class probabilities for some new data:

```python
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_iris

# Load the iris dataset
data = load_iris()
X, y = data.data, data.target

# Create a binary classification problem
X_binary = X[y != 2]  # Use only two classes for binary classification
y_binary = y[y != 2]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_binary, y_binary, test_size=0.2, random_state=42)

# Create a logistic regression model
model = LogisticRegression()

# Train the model on the training data
```

```
model.fit(X_train, y_train)

# Use the predict_proba method to get class probabilities
probabilities = model.predict_proba(X_test)

print(probabilities)
```

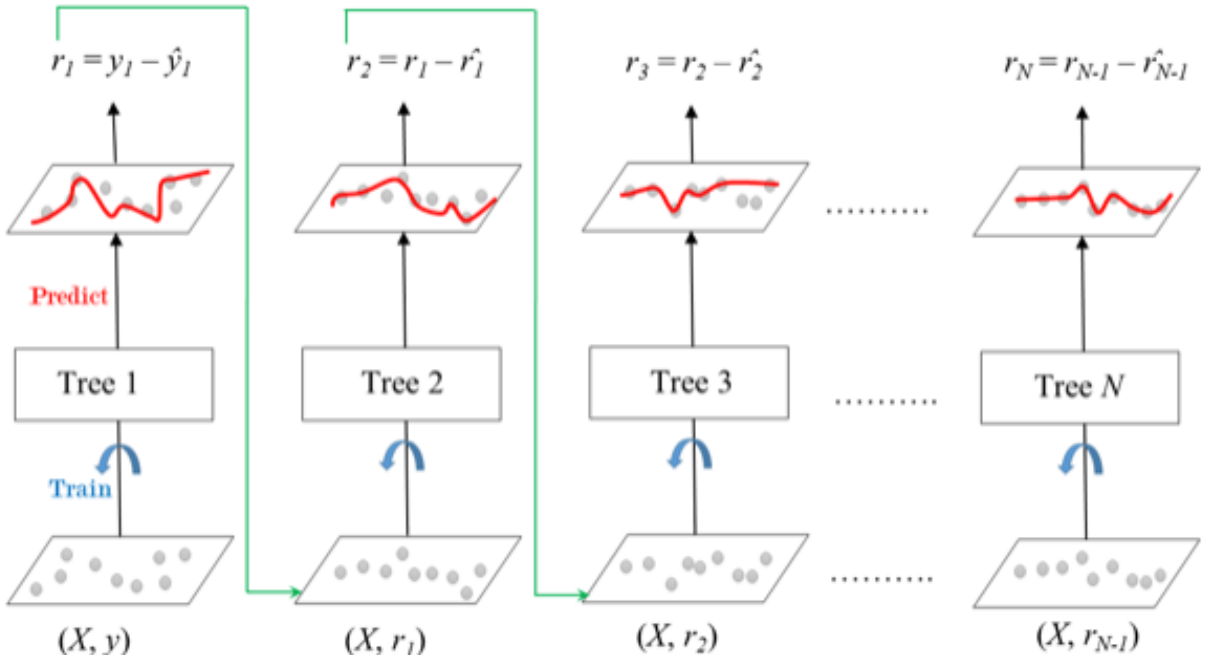
In this example, `probabilities` will be a 2D array containing the probability estimates for each class label. For binary classification, it will have two columns, where the first column represents the probability of the negative class, and the second column represents the probability of the positive class.

For multi-class classification tasks, the `.predict_proba()` method returns the probabilities for each class label. The shape of the output array will be `(n_samples, n_classes)`, where `n_samples` is the number of data points, and `n_classes` is the number of classes in the classification problem.

Having access to the predicted probabilities can be useful for various purposes, such as threshold tuning for binary classification, selecting the class with the highest probability for multi-class classification, or performing further analysis related to confidence in model predictions.

Gradient Boosting
also referred to as GB
a bit notorious for its proven record of winning many machine learning competitions
each predictor in the ensemble corrects its predecessor’s error
in contrast to AdaBoost, the weights of the training instances are not tweaked
instead each predictor is trained using the residual errors of its predecessor as ‘labels’
common technique called Gradient Boosted Trees, which uses a CART as a base learner

Training Gradient Boosted Trees for Regression
tree1 is trained using the features matrix X and the dataset labels y
predictions labeled y1hat are used to determine the training set residual errors called r1
tree2 is then trained using the features matrix X and the residual errors (r1) of tree1 as labels
predicted residuals r1hat are then used to determine the residuals of residuals which are labeled r2
visualize:



Shrinkage
an important parameter used in training gradient boosted trees
refers to the fact that the prediction of each tree in the ensemble is shrinked after it is multiplied by a learning rate (eta)
eta like AdaBoost is also range 0 to 1
same trade-off exists
the closer eta is to 1 the higher the chance for overfitting
*decreasing the eta needs to be compensated by increasing the number of estimators

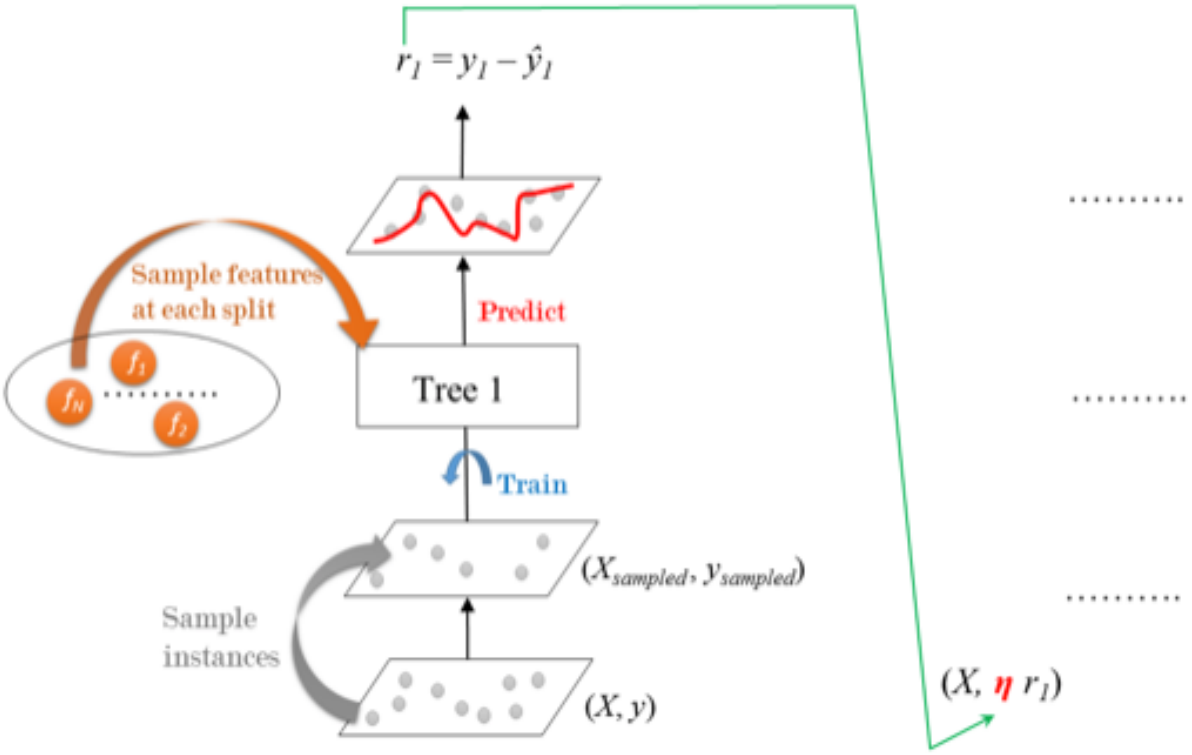
Prediction with GB Trees
Regression > ypred = y + eta(r1) + ... + eta(rn)
eta is represented as an exagerated lower case 'n' looking sign
class in scikit is GradientBoostingRegressor()
can also use classification with GB
scikit class GradientBoostingClassifier()

example
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error as MSE
SEED = 1
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=SEED)
#instantiate, key here is GradientBoostingRegressor contains arguments

n_estimators and max_depth
gbt = GradientBoostingRegressor(n_estimators=300, max_depth=1, random_state=SEED)
gbt.fit(X_train, y_train)
y_pred = gbt.predict(X_test)
rmse_test = MSE(y_test, y_pred)**(1/2)
print('Test set RMSE: {:.2f}'.format(rmse_test))

Cons of GB
involves an exhaustive search procedure
each CART is trained to find the best split points and features
this may lead to CARTs that use the same split-points and possibly the same features
to mitigate these effects you can use an algorithm known as stochastic gradient boosting

Stochastic Gradient Boosting
each CART is trained on a random subset of the training data
the sampled instances (40-80% of the training set) are sampled withou replacement
in addition, features are sampled (without replacement) when choosing split points (at the level of each node)
as a result, this creates further diversity in the ensemble and the net effect is adding more variance to the ensemble of trees
visualize:



only a fraction of the training instances are provided through sampling without replacement
during training not all features are considered when a split is made
once trained, predictions are made and the residual errors can be computed
the residual errors are multiplied by the eta and fed into the the next tree of the ensemble

example - use example above and plug this in
#instantiate GradientBoostRegressor with argument 'subsample' which tells each tree to sample 80% of the data for training and with argument 'max_features' which tells each tree to use 20% of available features to perform the best-split
sgbt = GradientBoostingRegressor(max_depth=1, subsample=0.8,
max_features=0.2, n_estimators=300, random_state=SEED)
sgbt.fit(X_train, y_train)
y_pred = sgbt.predict(X_test)
rmse_test = MSE(y_test, y_pred)**(1/2)
print('Test set RMSE: {:.2f}'.format(rmse_test))

Tuning a CART's Hyperparameters
ML models are characterized by parameters and hyperparameters
parameters are learned from data
ie split-point of a node, split-feature of a node, ...
hyperparameters are not learned from data
they should be set prior to training

ie max_depth, min_samples_leaf, splitting criterion, ...

Hyperparameter tuning
searching and setting optimal condition for the learning algorithm (model) to yield an optimal score
.score() function measures the agreement between true labels and a model's predictions
in scikit, it defaults to accuracy for classifiers and r-squared for regressors
cross validation is used to estimate the generalization performance

Approaches to hyperparameter tuning
-GridSearch
-Random Search
-Bayesian optimization
-genetic algorithms
-many more

Grid Search Cross Validation
-manually set a grid of discrete hyperparameter values
-pick a metric for scoring model performance
-search exhaustively through the grid
-for each set of hyperparameters, you evaluate each model's score
-optimal hyperparameters are those for which the model achieves the best cross-validation score
*Grid Search suffers from the curse of dimensionality (ie. the bigger the grid, the longer it takes to find the solution)

example
hyperparameter grid:
two dimensional (max_depth and min_samples_leaf)
max_depth = {2,3,4}
min_samples_leaf = {0.05, 0.1}
hyperparameter space = {(2,0.05), (2,0.1), ...}
for each combination of hyperparameters, the cross-validation score is evaluate using k-fold CV
*optimal hyperparameters = set of hyperparameters corresponding to the best CV score

```
from sklearn.tree import DecisionTreeClassifier
SEED = 1
dt = DecisionTreeClassifier(random_state=SEED)
#print out dt's hyperparameters using params() method
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'}
```

max_features is the number of feature to consider when looking for the best split
when max_features is a float, it is interpreted as a percentage
from sklearn.model_selection import GridSearchCV
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]}
grid_dt = GridSearchCV(estimator=dt, param_grid=params_dt, scoring='accuracy', cv=10, n_jobs=-1)
grid_dt.fit(X_train, y_train)
best_hyperparams = grid_dt.best_params_
print('Best hyperparameters:\n', best_hyperparams)
best_CV_score = grid_dt.best_score_
print('Best CV accuracy'.formtat(best_CV_score))
#extract best model from 'grid_dt'
best_model = grid_dt.best_estimator_
#note that this model is fitted on the whole training set because the refit parameter of GridSearchCV is set to True by default
#evaluate test set accuracy
test_acc = best_model.score(X_test, y_test)
print('Test set accuracy of best model: {:.3f}'.format(test_acc))

Sidebar
**if a dataset is imbalanced, use the ROC AUC score as a metric instead of accuracy

Random Forests Hyperparameters
RF is characterized by CART hyperparameters, number of estimators, bootstrap or not?, ...
tuning is computationally expensive
sometimes leads to very slight improvement
example

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error as MSE
SEED = 1
rf = RandomForestRegressor(random_state=SEED)
#to inspect rf's hyperparameters
rf.get_params()
#define a grid of hyperparameter 'params_rf' as a dictionary
#**dataset already loaded with train_test_split
param_rf = {'n_estimators': [200, 400, 500], 'max_depth': [4, 6, 8],
'min_samples_leaf': [0.1, 0.2], 'max_features': ['log2', 'sqrt']}
#instantiate
#'verbosity' argument controls the amount of messages printed during fitting
(higher the value, the more messages)
grid_rf = GridSearchCV(estimator=rf, param_grid= params_rf, cv=3,
scoring='neg_mean_squaredz_error', verbose=1, n_jobs=-1)
grid_rf.fit(X_train, y_train)
best_hyperparams = grid_rf.best_params_
print('Best hyperparameters:\n', best_hyperparams)
best_model = grid_rf.best_estimator_
y_pred = best_model.predict(X_test)
rmse_test = MSE(y_test, y_pred)**(1/2)
print('Test set RMSE of rf: {:.2f}'.format(rmse_test))
```