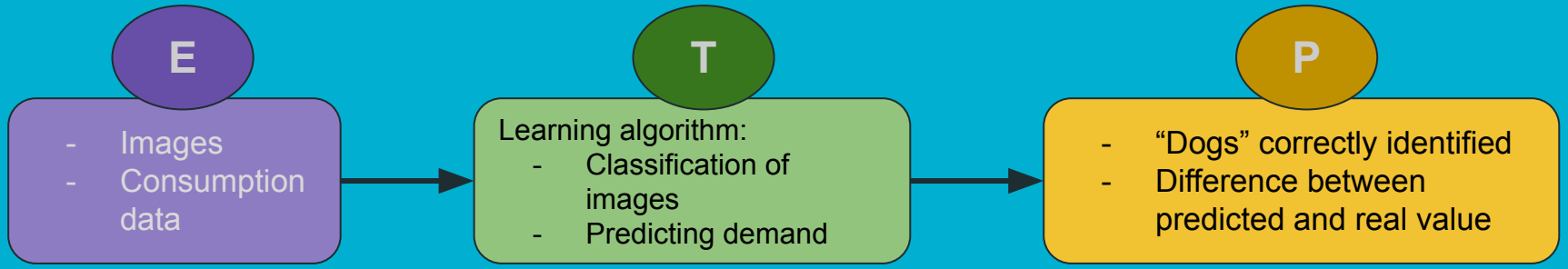
Week 3:

Decision Trees, Random Forests, and Bagging

Machine Learning

A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

T. Mitchell, Machine Learning (1997)



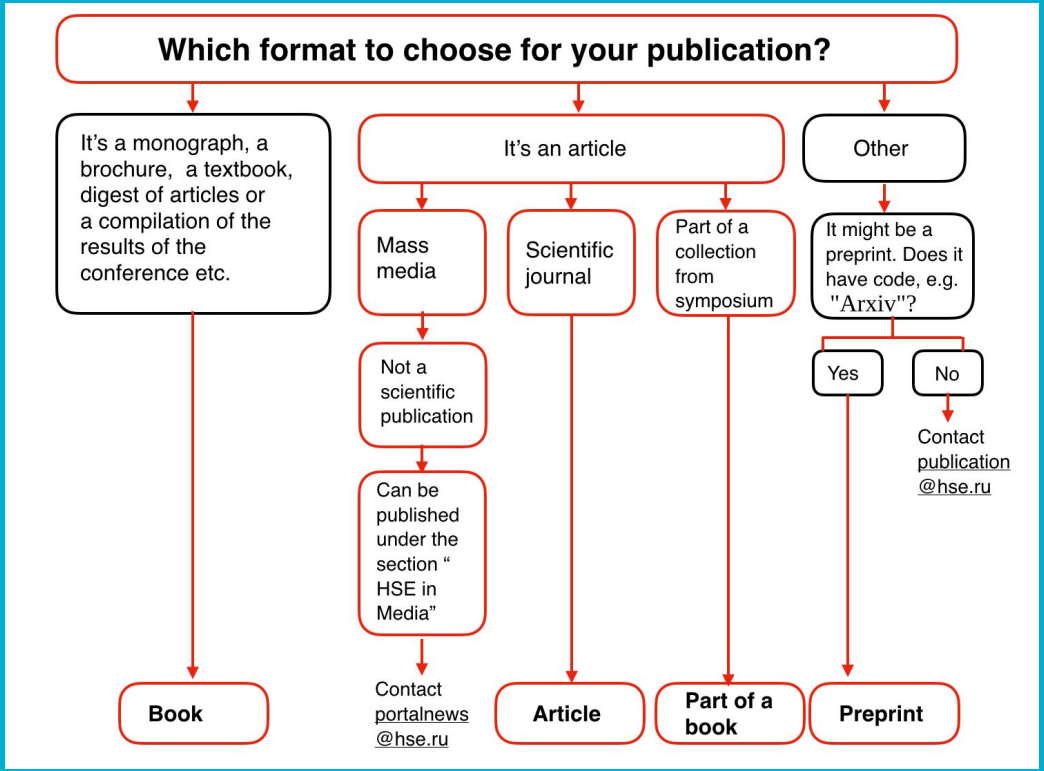
Decision Trees

Some of the most commonly used and intuitive algorithms

We commonly use decision trees in when making business decisions or just in everyday life

This example from the Higher School of Economics is a basic classifier that determines the proper media given the content

Analogous to simple if/else logic in programming



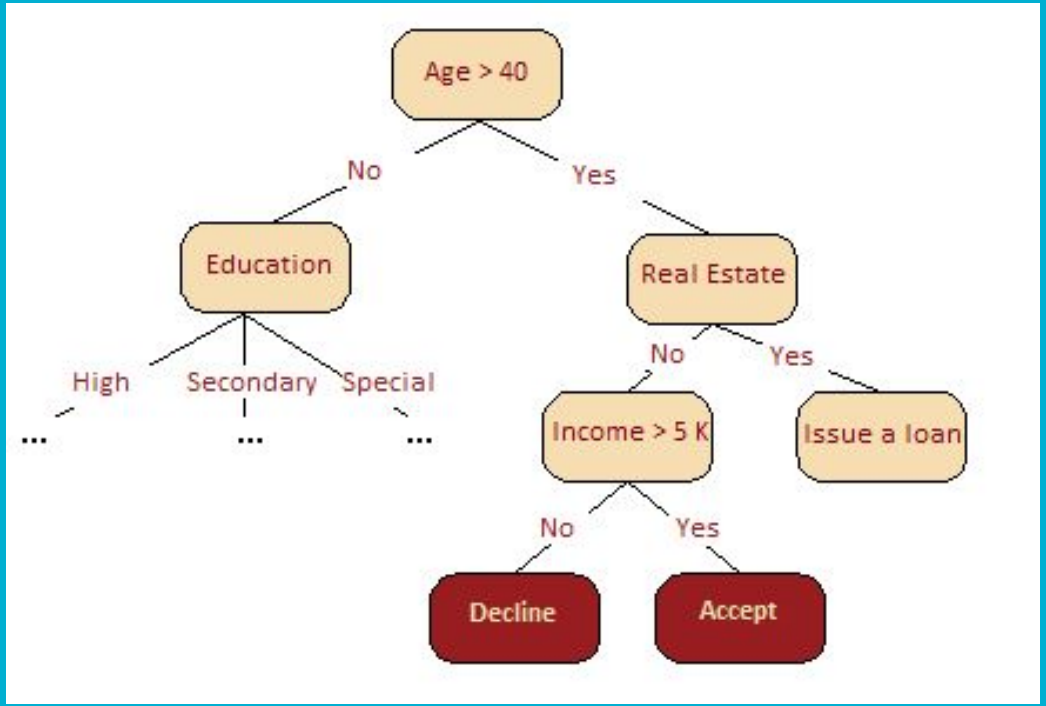
Decision Trees

Can be based on expert rules or heuristics

This example is a decision tree on whether to issue loans based on various socioeconomic indicators

The decision tree as a machine learning algorithm is essentially the same thing as the diagram shown

we incorporate a stream of logical rules of the form "feature *a* is less than *x* and feature *b* is less than *y* => Category 1" into a tree-like data structure



Decision Trees

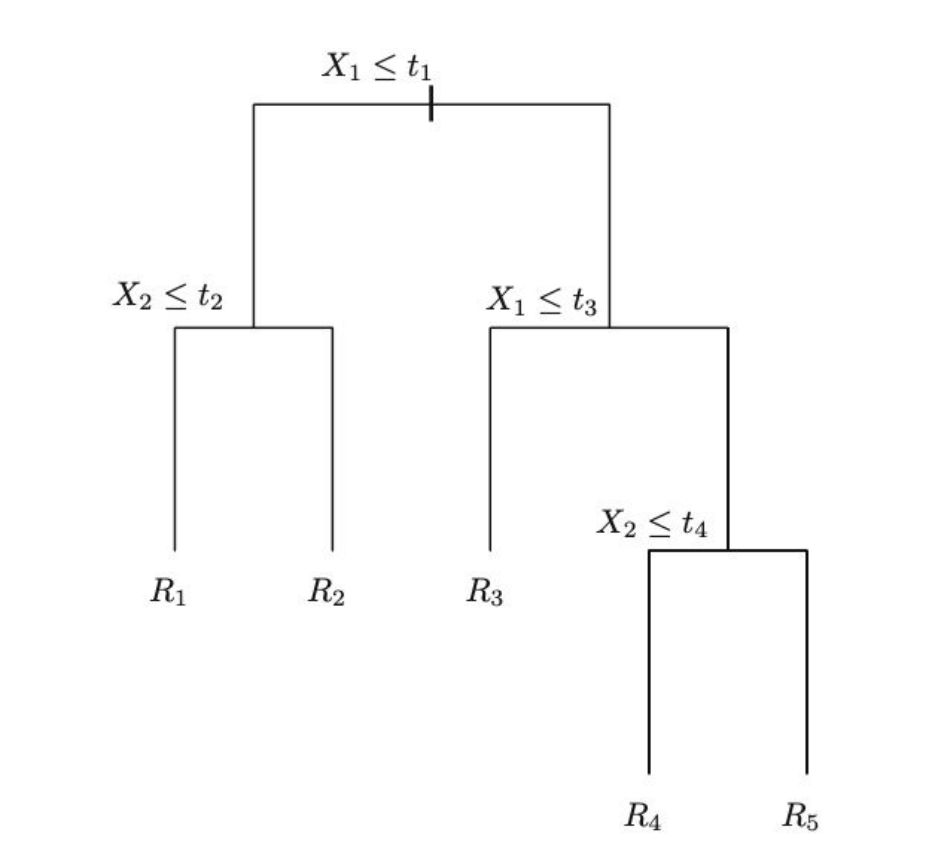
R1, R2, R3, R4 are “leaves”

X2<t2, X1<t3, X2<t4 are “nodes”

The connections between nodes and leaves are “branches”

Each level the tree grows is referred to as “depth”

All of these parameters can be tuned to prevent overfitting and minimize bias



Decision Trees

Decision tree classifiers utilize top-down recursive binary splitting to minimize entropy. Trees are considered “greedy” algorithms because they make the best decision at every particular node and don’t consider potential future decisions.

The basis of decision trees are well implemented in computer science through things like binary search algorithms, so the algorithms are common and well understood

Decision tree models can be a “black box” but algorithms exist to aid in explainability for complex models like m2cgen or Shapley values

Decision trees are some of the lightest-weight and fastest algorithms

Excellent for productionalization of models

Constructing Decision Trees

we saw that the decision to grant a loan is made based on age, assets, income, and other variables. But what variable to look at first? Let's discuss a simple example where all the variables are binary

Shannon's Entropy:

$$S = -\sum_{i=1}^{N} p_i \log_2 p_i$$

Gives us a metric of variability in the probability of a binary problem, where p is the probability

50/50 probability has maximum entropy *-(-0.5 - -0.5) = 1*

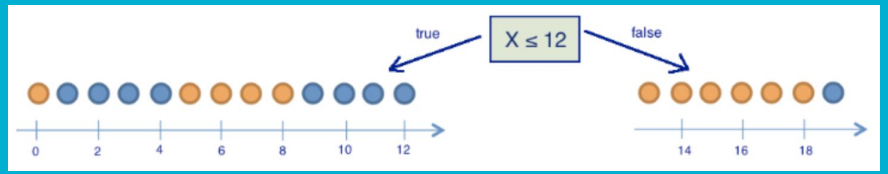
100/0 probability has minimum entropy *-(-0 - -0) = 0*

Uses $\log_2$ for referencing bits, but it's not really important

Constructing Decision Trees

To illustrate how entropy can help us identify good features for building a decision tree, let's look at a toy example. We will predict the color of the ball based on its position.

There are 9 blue balls and 11 yellow balls. If we randomly pull out a ball, then it will be blue with probability 9/20 and yellow with probability 11/20, which gives us an entropy: $S_0 = -\frac{9}{20}\log_2\frac{9}{20} - \frac{11}{20}\log_2\frac{11}{20} \approx 1$ This value by itself may not tell us much, but let's see how the value changes if we were to break the balls into two groups: with the position less than or equal to 12 and greater than 12.



The entropy of the left group is $S_1 = -\frac{5}{13}\log_2\frac{5}{13} - \frac{8}{13}\log_2\frac{8}{13} \approx 0.96$. The entropy of the right group is $S_2 = -\frac{1}{7}\log_2\frac{1}{7} - \frac{6}{7}\log_2\frac{6}{7} \approx 0.6$. As you can see, entropy has decreased in both groups, more so in the right group. Since entropy is, in fact, the degree of chaos (or uncertainty) in the system, the reduction in entropy is called information gain.

Constructing Decision Trees

$$IG(Q) = S_O - \sum_{i=1}^{q} \frac{N_i}{N} S_i,$$

where q is the number of groups after the split, N is number of objects from the sample in which variable Q is equal to the i-th value. In our example, our split yielded two groups (q=2), one with 13 elements (N_1=13), the other with 7 (N_2 = 7). Therefore, we can compute the information gain as:

$$IG(x \leq 12) = S_0 - \frac{13}{20} S_1 - \frac{7}{20} S_2 \approx 0.16.$$

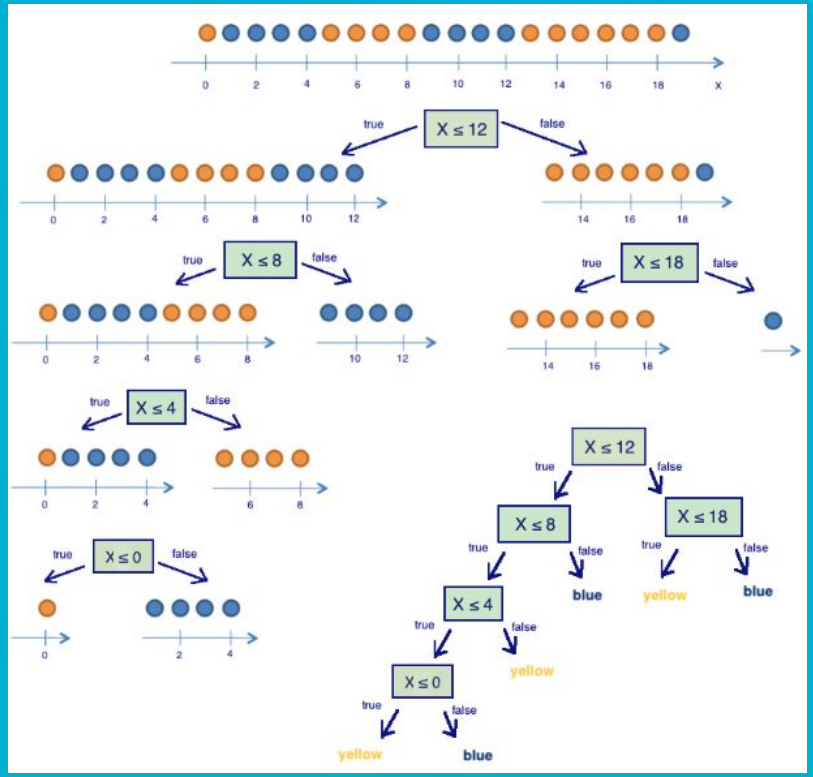
The intent when building a decision tree is to maximize information gain per split to make the tree as small as possible

Constructing Decision Trees

It turns out that dividing the balls into two groups by splitting on "coordinate is less than or equal to 12" gave us a more ordered system.

Let's continue to divide them into groups until the balls in each group are all of the same color.

it took only 5 "questions" (conditioned on the variable X) to perfectly fit a decision tree to the training set. Under other split conditions, the resulting tree would be deeper, i.e. take more "questions" to reach an answer.



Gini Uncertainty

An alternate splitting criterion from Shannon's entropy is Gini Uncertainty also called Gini Impurity.

The Gini impurity can be computed by summing the probability of an item with label being chosen times the probability of a mistake in categorizing that item. It reaches its minimum (zero) when all cases in the node fall into a single target category

Maximizing this criterion can be interpreted as the maximization of the number of pairs of objects of the same class that are in the same subtree

The equations of Shannon's Entropy and Gini Uncertainty for binary classification become:

$$S = -p_+\log_2 p_+ - p_-\log_2 p_- = -p_+\log_2 p_+ - (1-p_+)\log_2(1-p_+);$$

$$G = 1 - p_+^2 - p_-^2 = 1 - p_+^2 - (1-p_+)^2 = 2p_+(1-p_+).$$

Example

Let’s look at the code!

Decision Tree Regressors

Rather than applying entropy or uncertainty, with decision tree regressors we can apply the residual sum of squares (RSS): $\Sigma(y_i - f(x_i))^2$

Now finding the best binary partition in terms of minimum sum of squares is generally computationally infeasible. Hence we proceed with a greedy algorithm.

Starting with all of the data, consider a splitting variable j and split point s, and define the pair of half-planes:

$$R_1(j,s) = \{X|X_j \leq s\} \text{ and } R_2(j,s) = \{X|X_j > s\}.$$

Then we seek the splitting variable j and split point s that solve:

$$\min_{j,\ s}\left[\min_{c_1}\sum_{x_i \in R_1(j,s)}(y_i - c_1)^2 + \min_{c_2}\sum_{x_i \in R_2(j,s)}(y_i - c_2)^2\right].$$

(Hastie et al., 2017)

Decision Tree Regressors

For any choice j and s, the inner minimization is solved by:

$$\hat{c}_1 = \text{ave}(y_i | x_i \in R_1(j,s)) \quad \text{and} \quad \hat{c}_2 = \text{ave}(y_i | x_i \in R_2(j,s)).$$

For each splitting variable, the determination of the split point s can be done very quickly and hence by scanning through all of the inputs, determination of the best pair (j, s) is feasible.

Having found the best split, we partition the data into the two resulting regions and repeat the splitting process on each of the two regions. Then this process is repeated on all of the resulting regions.

(Hastie et al., 2017)

Regression Example

Let’s look at the code!

Even the example tends to overfit!

Crucial Tree Parameters

Technically, you can build a decision tree until each leaf has exactly one instance, but the tree would be WAY overfit.

This will result in trees that partition on meaningless features or noise just to maximize information gain

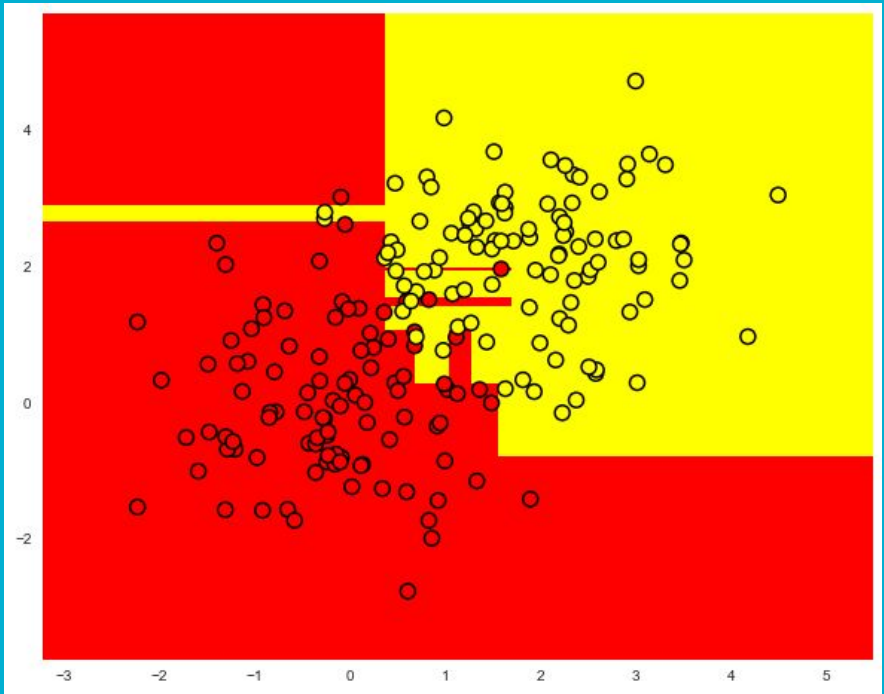
e.g. Colour of the loan applicant's pants; first letter of the baseball pitcher's name; DLS section number…

EXCEPT:

Random Forests → average the response of multiple "overfitted" trees

Pruning → building trees to the bottom and then "pruning" nodes that have little improvement on model performance

Overfit Trees



The most common ways to deal with overfitting in decision trees:

- artificial limitation of the depth or a minimum number of samples in the leaves. Then the construction of a tree just stops at some point;
- pruning the tree.

Limiting Tree Depth

Tree size is a tuning parameter governing the model's complexity, and the optimal tree size should be adaptively chosen from the data.

One approach would be to split tree nodes only if the decrease in sum-of-squares due to the split exceeds some threshold. sk-learn parameter examples would be *min_impurity_decrease* or *min_impurity_split*

You can also limit tree growth by the maximum depth or maximum number of nodes

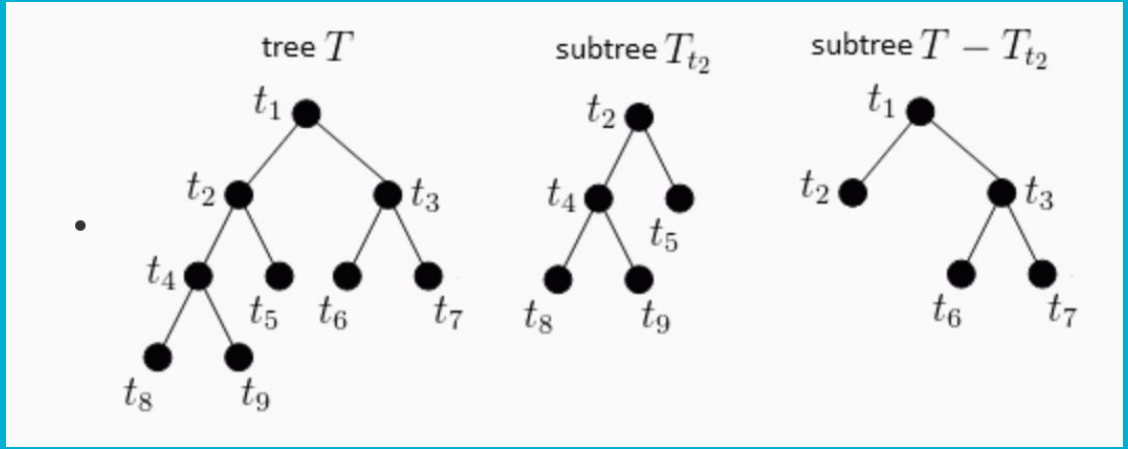
(Hastie et al., 2017)

Tree Pruning

Rather than limiting the growth of a tree, it's often more advisable to let a tree grow to a very large depth, and then "prune" the nodes and subsequent subtrees which do not add significant reduction of residuals

Cost complexity pruning (ccp) is a built into the sk-learn decision tree API

Simply remove a subtree and measure the difference in the residuals. If the difference is negative, removal of the subtree improves model bias.



(Hastie et al., 2017, Breiman et al., 1984)

Tree Pruning

we define the cost complexity criterion:

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha|T|.$$

The tuning parameter α ≥ 0 governs the trade off between tree size and its goodness of fit to the data.

Large values of α result in smaller trees Tα, and conversely for smaller values of α.

For each value of α one can show that there is a unique smallest subtree Tα that minimizes $C_\alpha(T)$

As the notation suggests, with α = 0 the solution is the full tree T_0

(Hastie et al., 2017, Breiman et al., 1984)

Classification Trees in sk-learn

The main parameters of the sklearn.tree.DecisionTreeClassifier class are:

- max_depth – the maximum depth of the tree;
- max_features - the maximum number of features with which to search for the best partition (this is necessary with a large number of features because it would be "expensive" to search for partitions for *all* features);
- min_samples_leaf – the minimum number of samples in a leaf. This parameter prevents creating trees where any leaf would have only a few members.

The parameters of the tree need to be set depending on input data, and it is usually done by means of *cross-validation*.

Ensemble Methods

An *ensemble* is a set of elements that collectively contribute to a whole.

Condorcet's jury theorem (1784) is about an ensemble in some sense. It states that, if each member of the jury makes an independent judgement and the probability of the correct decision by each juror is more than 0.5, then the probability of the correct decision by the whole jury increases with the total number of jurors and tends to one. On the other hand, if the probability of being right is less than 0.5 for each juror, then the probability of the correct decision by the whole jury decreases with the number of jurors and tends to zero.

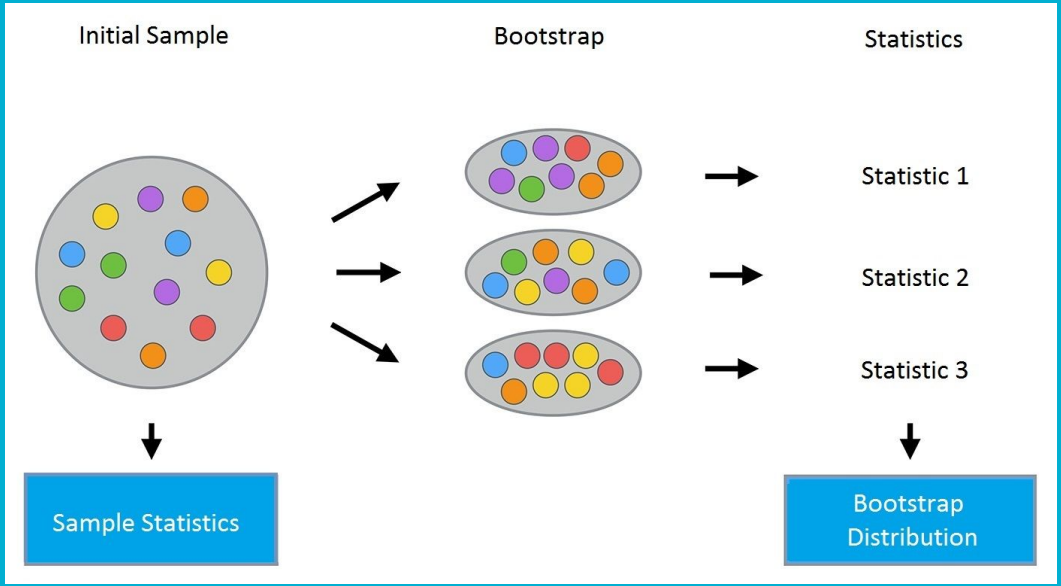
In ML terms, if the accuracy of each model in the ensemble is greater than 50% by some metric, then aggregating several models together will increase the accuracy of the total model.

Likewise, if the accuracy of each individual model is less than 50%, then the ensemble model will reduce the accuracy.

See also: Galton’s Wisdom of the Crowd (1906)

Bootstrap Aggregation (Bagging)

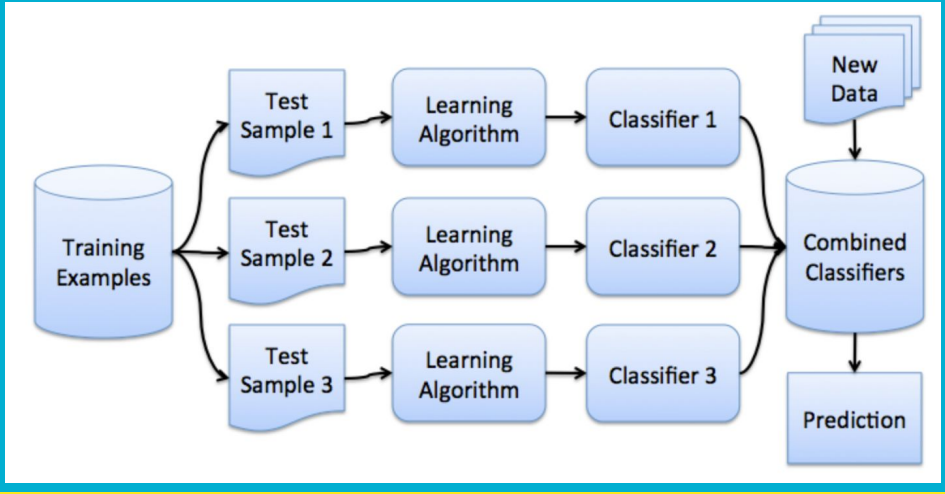
One of the first and most basic ensemble techniques. It was proposed by Leo Breiman in 1994. Bagging is based on the statistical method of bootstrapping, which makes the evaluation of many statistics of complex models feasible.



Bootstrap Aggregation (Bagging)

Suppose that we have a training set X. Using bootstrapping, we generate samples X_1, X_2, … X_M. Now, for each bootstrap sample, we train its own classifier $a_i(x)$. The final classifier will average the outputs from all these individual classifiers. In the case of classification, this technique corresponds to voting:

$$a(x) = \frac{1}{M}\sum_{i=1}^{M} a_i(x).$$



Bootstrap Aggregation (Bagging)

Let's consider a regression problem with base algorithms $b_1(x)$, … $b_n(x)$. Assume that there exists an ideal target function $y(x)$ of true answers defined for all inputs and that the distribution $p(x)$ is defined. We can then express the error for each regression function as follows:

$$\varepsilon_i(x) = b_i(x) - y(x), \quad i = 1, \ldots, n$$

And the expected value of the mean squared error:

$$\mathrm{E}_x\left[(b_i(x) - y(x))^2\right] = \mathrm{E}_x\left[\varepsilon_i^2(x)\right].$$

Then, the mean error over all regression functions will look as follows:

$$\mathrm{E}_1 = \frac{1}{n}\mathrm{E}_x\left[\sum_i^n \varepsilon_i^2(x)\right]$$

Now, let's construct a new regression function that will average the values from the individual functions:

$$a(x) = \frac{1}{n}\sum_{i=1}^{n} b_i(x)$$

Bootstrap Aggregation (Bagging)

Let's find its mean squared error:

$$
\begin{aligned}
E_n &= E_x\left[\frac{1}{n}\sum_{i=1}^{n} b_i(x) - y(x)\right]^2 \\
&= E_x\left[\frac{1}{n}\sum_{i=1}^{n} \varepsilon_i\right]^2 \\
&= \frac{1}{n^2} E_x\left[\sum_{i=1}^{n} \varepsilon_i^2(x) + \sum_{i \neq j} \varepsilon_i(x)\varepsilon_j(x)\right] \\
&= \frac{1}{n} E_1
\end{aligned}
$$

Thus, by averaging the individual answers, we reduced the mean squared error by a factor of n.

Bagging reduces the variance of a classifier by decreasing the difference in error when we train the model on different datasets. In other words, **bagging prevents overfitting**. The efficiency of bagging comes from the fact that the individual models are quite different due to the different training data and their **errors cancel each other out** during voting. Additionally, **outliers are likely omitted** in some of the training bootstrap samples.

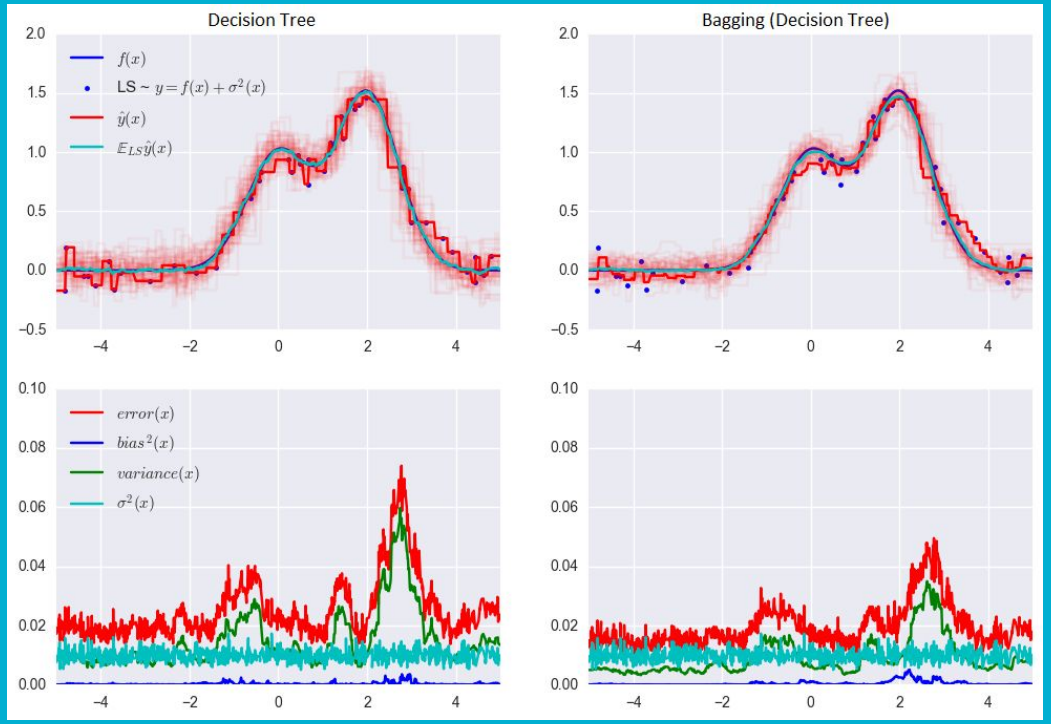
Bootstrap Aggregation (Bagging)

The scikit-learn library supports bagging with meta-estimators BaggingRegressor and BaggingClassifier. You can use most of the algorithms as a base.

Let's examine how bagging works in practice and compare it with a decision tree. For this, we will use an example from sk-learn's documentation.

As you can see from the graph to the right, the variance in the error is much lower for bagging. Remember that we have already proved this theoretically.

Bagging is effective on small datasets. Dropping even a small part of training data leads to constructing substantially different base classifiers. If you have a large dataset, you would generate bootstrap samples of a much smaller size.



Building Bagged Trees

To apply bagging to regression trees, we simply construct B regression trees using B bootstrapped training sets, and average the resulting predictions.

These trees are grown deep, and are not pruned. Hence each individual tree has high variance, but low bias. Averaging these B trees reduces the variance.

Bagging has been demonstrated to give impressive improvements in accuracy by combining together hundreds or even thousands of trees into a single procedure

Bootstrap Aggregation (Bagging)

The previous example is unlikely to be applicable to any real work. This is because we made a strong assumption that our individual errors are independent and identically distributed (IID).

More often than not, this is way too optimistic for real-world applications. When this assumption is false, the reduction in error will not be as significant.

In the following lectures, we will discuss some more sophisticated ensemble methods, which enable more accurate predictions in real-world problems.

Out of Bag Error (OOB)

With Random Forests there is no *explicit* need to use cross-validation or hold-out samples in order to get an unbiased error estimation. Why? Because, in ensemble techniques, the error estimation takes place internally.

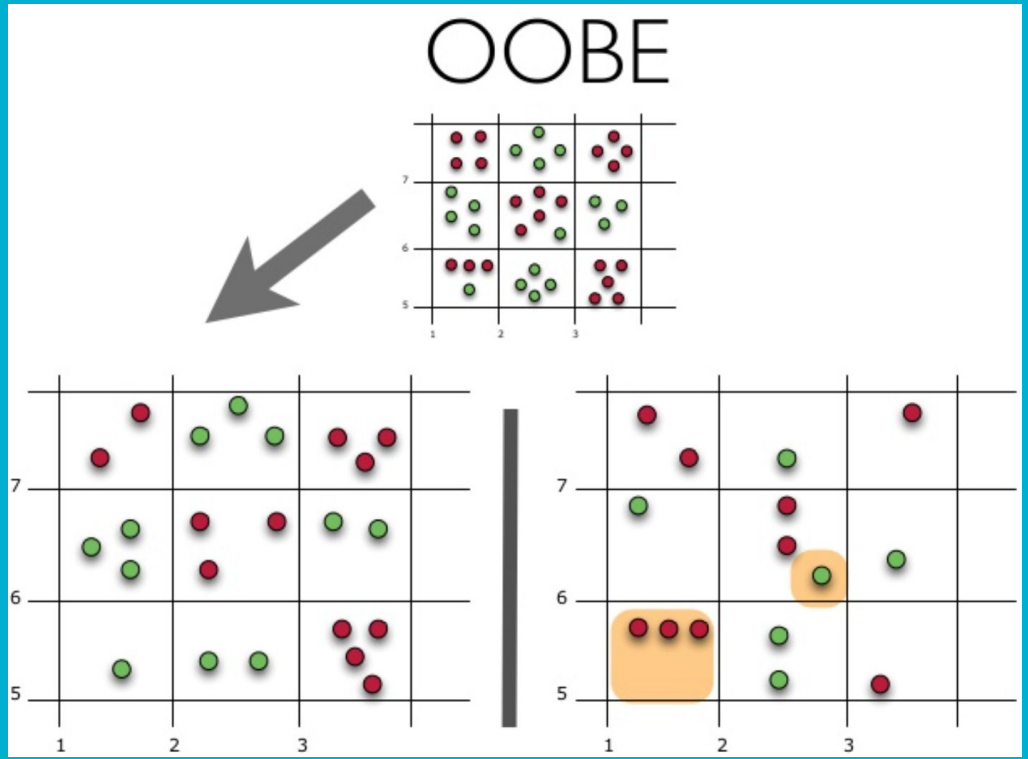
Random trees are constructed using different bootstrap samples of the original dataset. Approximately 37% of inputs are left out of a particular bootstrap sample and are not used in the construction of the k-th tree.

This is easy to prove. Suppose there are ℓ examples in our dataset. At each step, each data point has equal probability of ending up in a bootstrap sample with replacement, probability $1/\ell$ The probability that there is no such bootstrap sample that contains a particular dataset element (i.e. it has been omitted ℓ times) equals $(1-1/\ell)^{\ell}$. When $\ell \to \pm\infty$, it becomes equal to the Second Remarkable Limit 1/e. Then, the probability of selecting a specific example is 1-1/e = 63%.

Out of Bag Error (OOB)

The top part of the figure above represents our original dataset. We split it into the training (left) and test (right) sets. In the left image, we draw a grid that perfectly divides our dataset according to classes. Now, we use the same grid to estimate the share of the correct answers on our test set. We can see that our classifier gave incorrect answers in those 4 cases that have not been used during training (on the left). Hence, the accuracy of our classifier is 11/15 = 73.33%

To sum up, each base algorithm is trained on 63% of the original examples. It can be validated on the remaining 37%. The Out-of-Bag estimate is nothing more than the mean estimate of the base algorithms on those 37% of inputs that were left out of training.



Random Forests

The problem with bagged trees is that the first few splits will likely be on the same features so most of the trees will look very similar, and averaging similar trees doesn't provide much benefit for error reduction. Random forests provide an improvement over bagged trees by forcing decorrelation of the trees with a random selection of features at each split.

When building the trees, each time a split in a tree is considered, a random sample of m predictors is chosen as split candidates from the full set of p predictors. The split is allowed to use only one of those m predictors. A fresh sample of m predictors is taken at each split, and typically we choose $m \approx \sqrt{p}$—that is, the number of predictors considered at each split is approximately equal to the square root of the total number of predictors

In other words, in building a random forest, at each split in the tree, the algorithm is not even allowed to consider a majority of the available predictors.

Using a small value of m in building a random forest will typically be helpful when we have a large number of correlated predictors

(James et al., 2013)

Random Forests

Let’s look at the code!

Variance and Decorrelation

One major problem with trees is their high variance. Often a small change in the data can result in a very different series of splits, making interpretation somewhat precarious.

The major reason for this instability is the hierarchical nature of the process: the effect of an error in the top split is propagated down to all of the splits below it.

One can alleviate this to some degree by trying to use a more stable split criterion, but the inherent instability is not removed. It is the price to be paid for estimating a simple, tree-based structure from the data.

Variance and Decorrelation

Let's write the variance of a random forest as: $\rho(x) = \text{Corr}\left[T(x_1, \Theta_1(Z)), T(x_2, \Theta_2(Z))\right],$

where:

$\rho(x)$ is the sample correlation coefficient between any two trees used in averaging
$\Theta_1(Z)$ and $\Theta_2(Z)$ are a randomly selected pair of trees on randomly selected elements of the sample
$T(x,\Theta_i(Z)$ is the output of the i-th tree classifier on an input vector

$\sigma^2(x)$ is the sample variance of any randomly selected tree: $\sigma^2(x) = \text{Var}\left[T(x, \Theta(X))\right]$

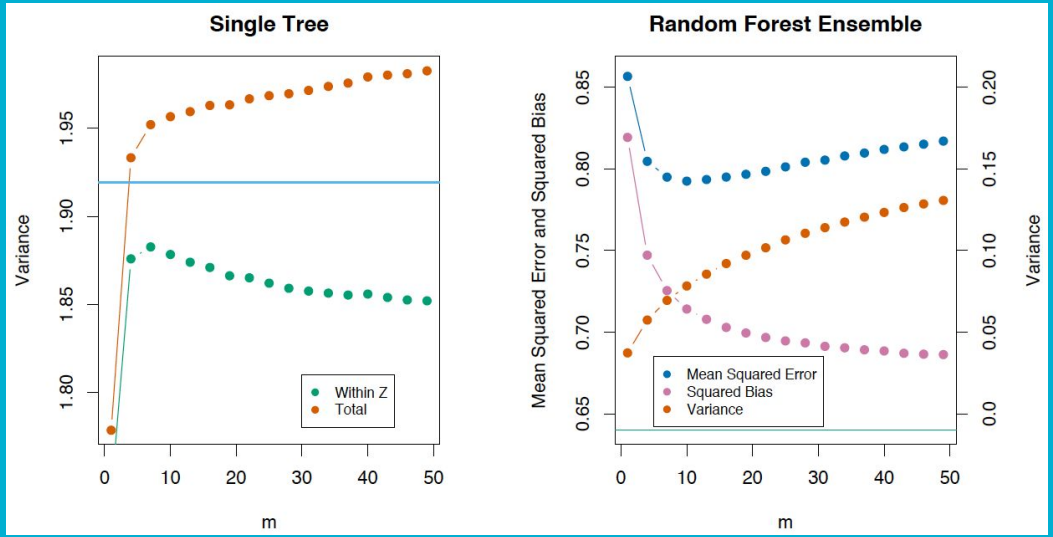
It is easy to confuse $\rho(x)$ with the average correlation between the trained trees in a given random forest when we consider trees as N-vectors and calculate the average pairwise correlation between them. But this is not the case.

correlation is not directly related to the averaging process, and the dependence of $\rho(x)$ on x warns us of this difference. $\rho(x)$ is the theoretical correlation between a pair of random trees estimated on the input x. Its value comes from the repeated sampling of the training set from the population Z and the subsequent random choice of a pair of trees. In statistics jargon, this is the correlation caused by the sampling distribution of Z and Θ.

Variance and Decorrelation

The conditional covariance of any pair of trees is equal to 0 if bootstrapping and feature selection are independent and identically distributed.

Let's consider the variance of a single tree versus an ensemble of trees. The variance of the single tree is much higher.



(Hastie et al., 2017)

Bias

Just as in bagging, the bias of a random forest is the same as the bias of a single tree $T(x,\Theta_i(Z)$:

$$\begin{aligned} \text{Bias} &= \mu(x) - \mathrm{E}_Z\, f_{rf}(x) \\ &= \mu(x) - \mathrm{E}_Z\, \mathrm{E}_{\Theta|Z}\, T(x, \Theta(Z)) \end{aligned}$$

In absolute value, the bias is usually higher than that of an unpruned tree because randomization and sample space reduction impose their own restrictions on the model.

Therefore, the improvements in prediction accuracy obtained by bagging and random forests are solely the result of variance reduction!

Extremely Random Trees

Extremely Randomized Trees employ a greater degree of randomization at the cut-point choice when splitting a tree node. As in random forests, a random subset of features is used.

But, instead of the search for the optimal thresholds, their values are selected at random for each possible feature, and the best one among these randomly generated thresholds is used as the best rule to split the node.

This usually trades off a slight reduction in the model variance with a small increase of the bias.

Extra random trees should be used if you have greatly overfit with random forests or gradient boosting.

Random Forests for Unsupervised Learning

Using the scikit-learn method RandomTreesEmbedding, we can transform our dataset into a high-dimensional, sparse representation. We first build extremely randomized trees and then use the index of the leaf containing the example as a new feature.

For example, if the input appears in the first leaf, we assign 1 as the feature value; if not, we assign 0.

We can control the number of features and the sparseness of data by increasing or decreasing the number of trees and their depth. Because nearby data points are likely to fall into the same leaf, this transformation provides an implicit nonparametric estimate of their density.

Random Forest Pros

- High prediction accuracy; will perform better than linear algorithms in most problems; the accuracy is comparable with that of boosting.
- Robust to outliers, thanks to random sampling.
- Insensitive to the scaling of features as well as any other monotonic transformations due to the random subspace selection.
- Doesn't require fine-grained parameter tuning, works quite well out-of-the-box. With tuning, it is possible to achieve a 0.5–3% gain in accuracy, depending on the problem setting and data.
- Efficient for datasets with a large number of features and classes.
- Handles both continuous and discrete variables equally well.

Random Forest Pros

- Rarely overfits. In practice, an increase in the tree number almost always improves the composition. But, after reaching a certain number of trees, the learning curve is very close to the asymptote.
- There are developed methods to estimate feature importance.
- Works well with missing data and maintains good accuracy even when a large part of data is missing.
- Provides means to weight classes on the whole dataset as well as for each tree sample.
- Under the hood, calculates proximities between pairs of examples that can subsequently be used in clustering, outlier detection, or interesting data representations.
- The above functionality and properties may be extended to unlabeled data to enable unsupervised clustering, data visualization, and outlier detection.
- Easily parallelized and highly scalable.

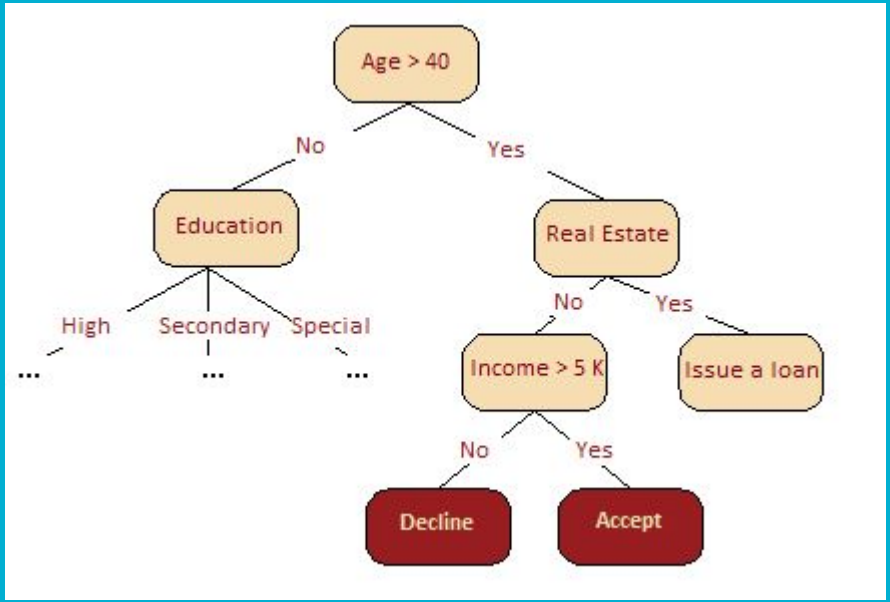
Random Forest Cons

- In comparison with a single decision tree, a Random Forest's output is more difficult to interpret.
- There are no formal p-values for feature significance estimation.
- Performs worse than linear methods in the case of sparse data: text inputs, bag of words, etc.
- Unlike linear regression, Random Forest is unable to extrapolate. But, this can be also regarded as an advantage because outliers do not cause extreme values in Random Forests.

Random Forest Cons

- Prone to overfitting in some problems, especially, when dealing with noisy data.
- In the case of categorical variables with varying level numbers, random forests favor variables with a greater number of levels. The tree will fit more towards a feature with many levels because this gains greater accuracy.
- If a dataset contains groups of correlated features, preference might be given to groups of smaller size.
- The resulting model is large and requires a lot of RAM.

Feature Importance



From the picture to the left, it is intuitively clear that, in our credit scoring problem, *Age* is much more important than *Income*. This can be formally explained using the concept of *information gain*.

In the case of many decision trees or a random forest, the closer the mean position of a feature over all the trees to the root, the more significant it is for a given classification or regression problem.

There are a lot of methods to assess feature importances. Leo Breinman in his works suggested to evaluate the importance of a variable by measuring decrease of accuracy of the forest when the variable is randomly permuted or decrease of impurity of a nodes where the given variable is used for splitting. The former method is often called **permutation importance**. The latter method is used in sklearn.

Feature Importance

The average reduction in accuracy caused by a variable is determined during the calculation of the out-of-bag error. The greater the reduction in accuracy due to an exclusion or permutation of the variable, the higher its *importance score*. For this reason, variables with a greater average reduction in accuracy are generally more significant for classification.

Sklearn library uses another approach to determine feature importance. The rationale for that method is that the more gain in information the node (with splitting feature X_j) provides, the higher its importance.

The average reduction in the Gini impurity – or MSE for regression – represents the contribution of each feature to the homogeneity of nodes and leaves in the resulting Random Forest model. Each time a selected feature is used for splitting, the Gini impurity of the child nodes is calculated and compared with that of the original node.

Gini impurity is a score of homogeneity with the range from 0 (homogeneous) to 1 (heterogeneous). The changes in the value of the splitting criterion are accumulated for each feature and normalized at the end of the calculation. A higher reduction in the Gini impurity signals that splitting results by this feature results in nodes with higher purity

TreeSHAP

SHAP (Shapley Additive Explanations) utilize concepts from game theory to calculate the contribution of every feature to the model.

TreeSHAP is a variant of SHAP for tree-based machine learning models, but can sometimes produce misleading results with correlated features. If a unimportant feature is unimportant, but correlated with an important feature it can be assigned a non-zero importance.

SHAP values are very useful for communicating the feature importance of a complex model. Can be an important product in of themselves!

References

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An introduction to statistical learning* (Vol. 112, p. 18). New York: springer.

https://github.com/Yorko/mlcourse.ai/tree/master/jupyter_english/topic03_decision_trees_kNN

https://github.com/Yorko/mlcourse.ai/tree/master/jupyter_english/topic05_ensembles_random_forests