Trees and Ensembles

Jonas Nordqvist

jonas.nordqvist@lnu.se

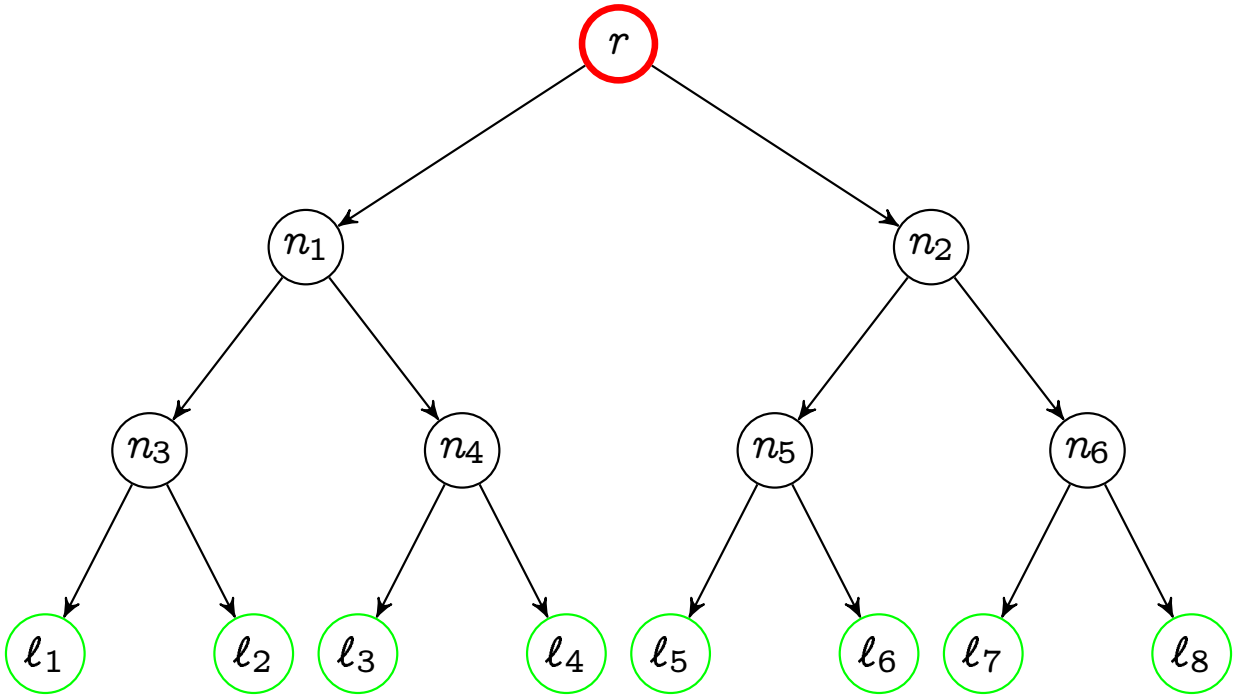
Agenda

- Decision trees – an example
- Building a regression tree
- How to avoid overfitting
  - Depth control
  - Pruning
- Classification trees
- Decision trees in `sklearn`
- Ensembles and Random forests

Reading Instructions
Chapter 2 and 7

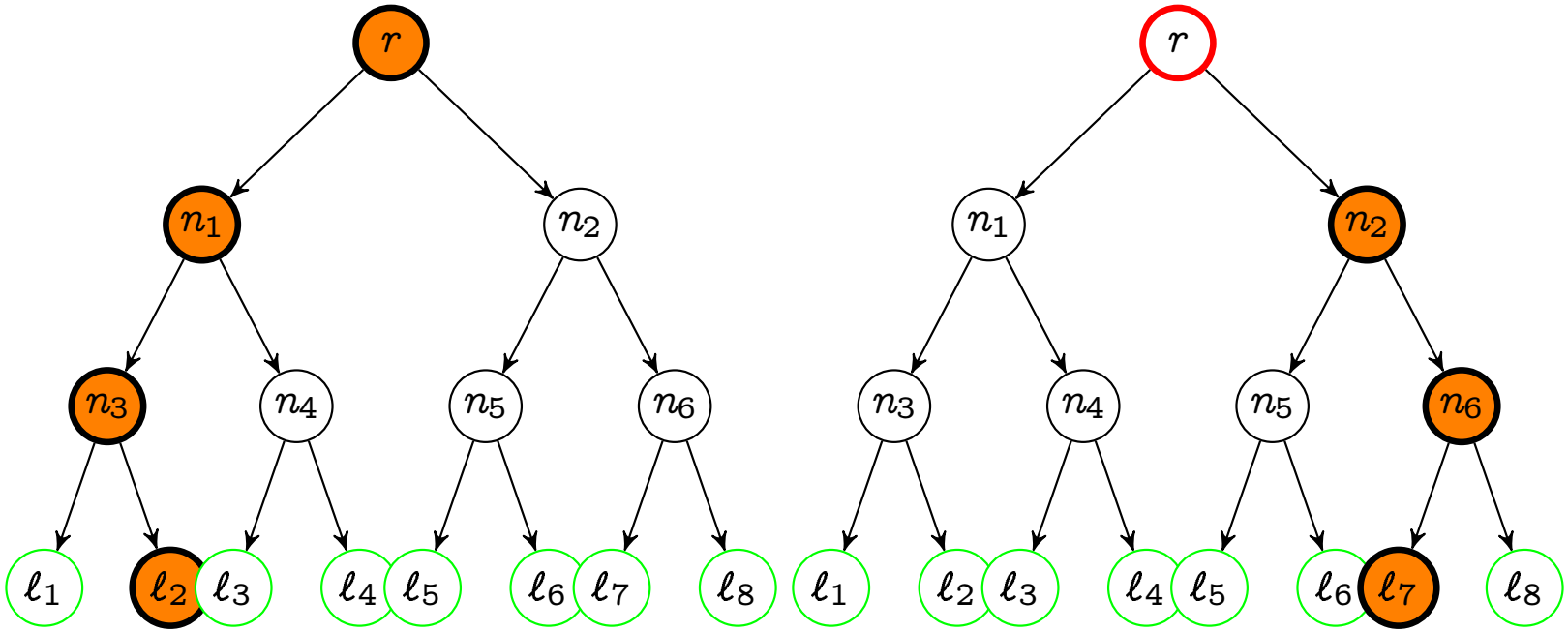
Properties of trees

A *tree* is basic data structure, consisting of a set of nodes and a set of directed edges. There are two special types of nodes the *root* r and the *leafs* ℓ_i. Every node (except the root) has a parent node. Parents and children nodes are adjoined by an edge.



Properties of trees

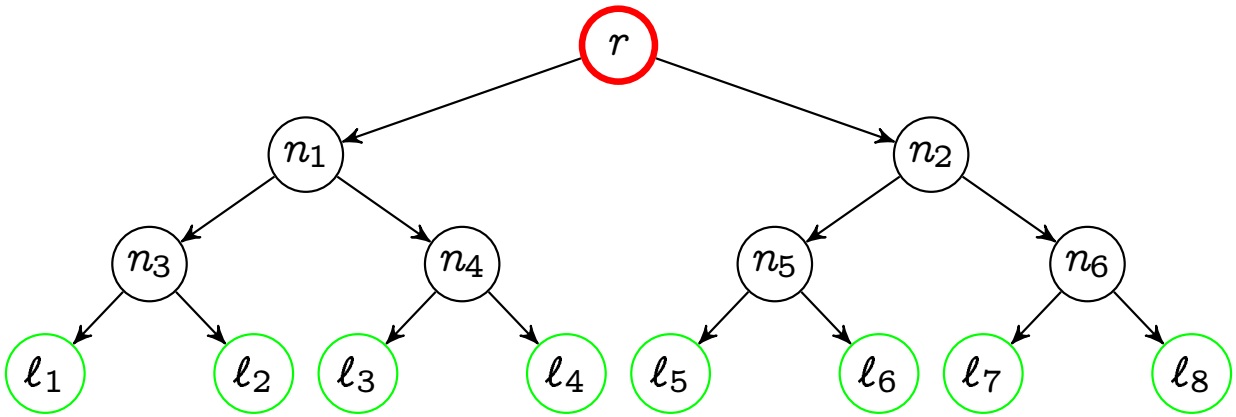
A *path* in a tree is a sequence of of nodes and edges adjoining a node and one of its descendants. Below are two examples of paths, from the root to the leaf ℓ_2 and from the node n_2 to the leaf ℓ_7.



Properties of trees

Any node of a tree has the following properties

1. The *depth* of a node is the number of edges between the node and the root node. The root has depth 0.
2. The *height* of a node is the number of edges on the longest path from the node to a leaf. Any leaf has height 0.



Note that these trees are *binary trees* (*i.e.* only two children per node), but this is not a necessary property of a tree.

We start by an example

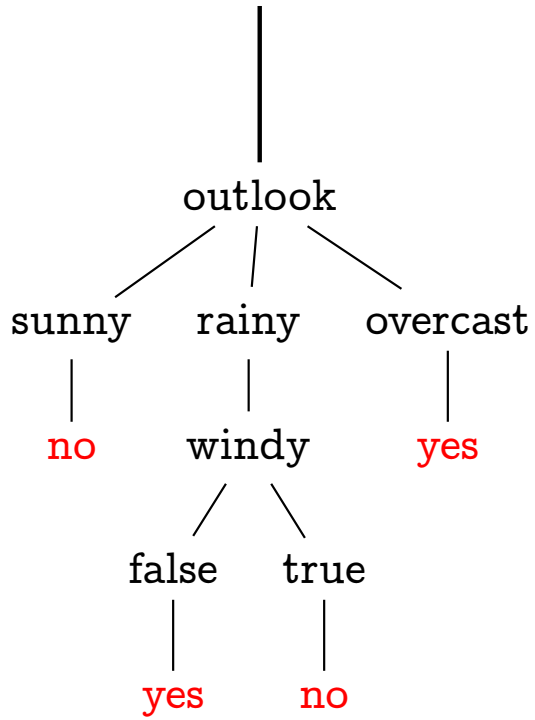
Can we find a model which decides whether it is a good idea to go out and play or not?

outlook	temp.	humidity	windy	play
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cool	normal	false	yes
rainy	cool	normal	true	no
overcast	cool	normal	true	yes

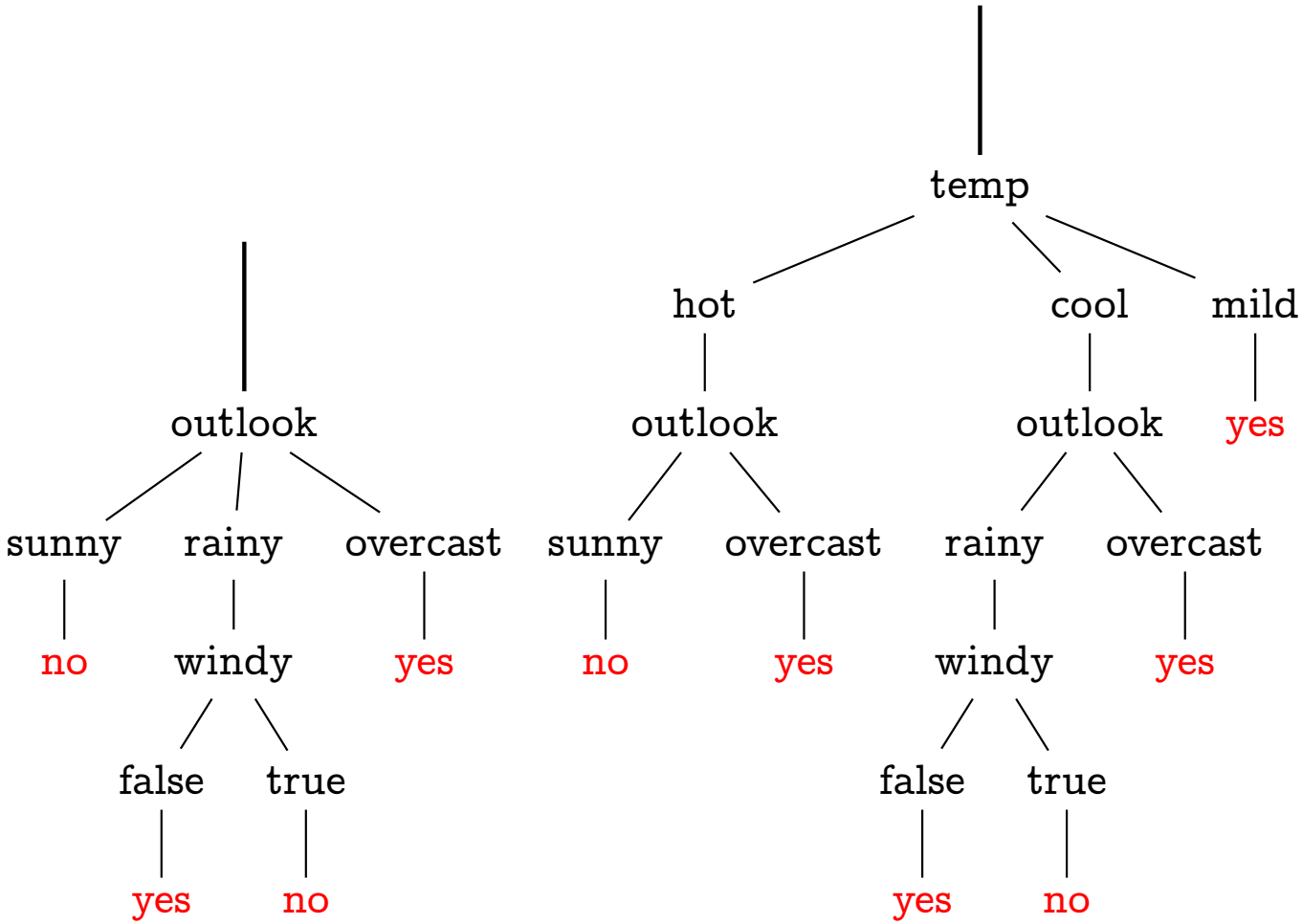
Decision tree

Decision trees are trees in which nodes represent features and its edges represent different values for the feature. In the right picture below, we can think of the root node as the question 'What is the outlook?', and its edges as the answers to said question. Leafs represent the predictions.

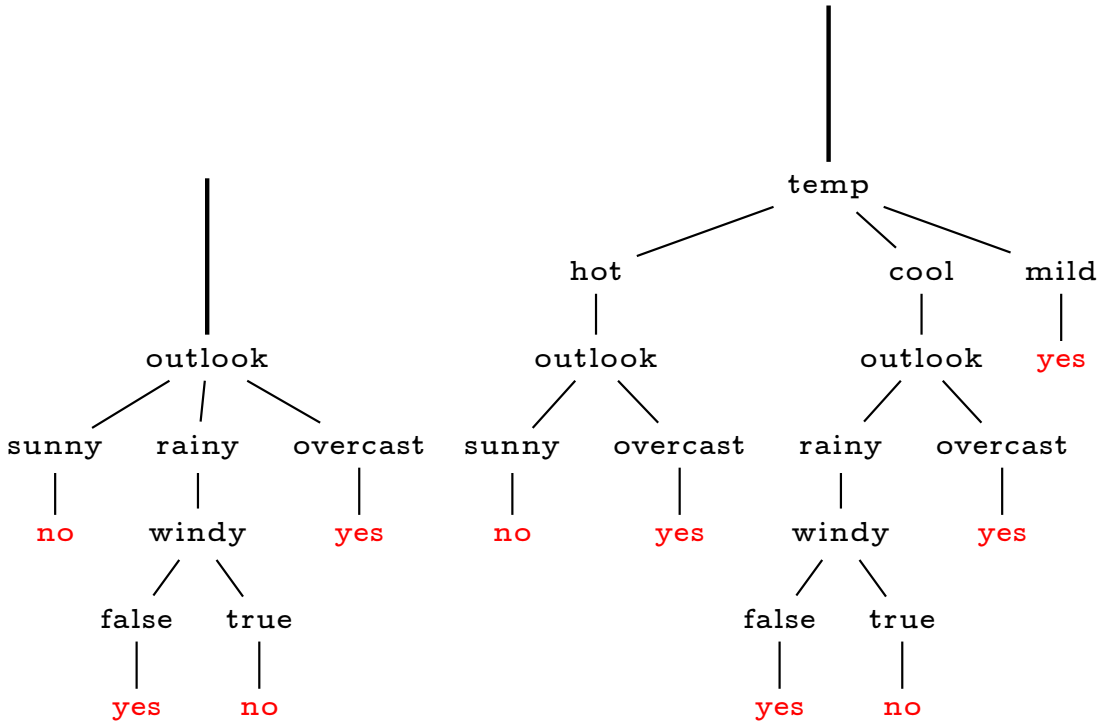
outlook	temp.	humidity	windy	play
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cool	normal	false	yes
rainy	cool	normal	true	no
overcast	cool	normal	true	yes



Two comparable trees



Which tree is preferred?



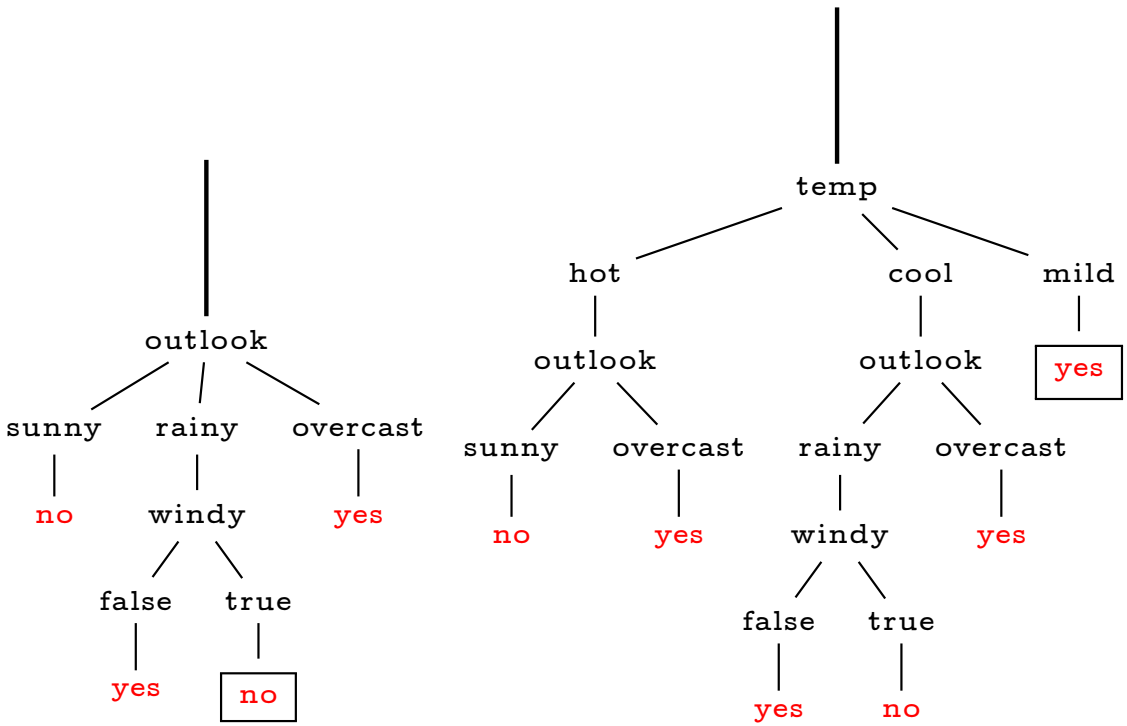
Both trees solves the problem of describing the data, which is preferred?

Finding the smallest possible tree is shown to be NP-hard. Hence, in order to construct our decision trees we will use greedy methods.

Unseen examples

What happens to new samples?

outlook	temp.	humidity	windy	play
rainy	mild	high	true	no

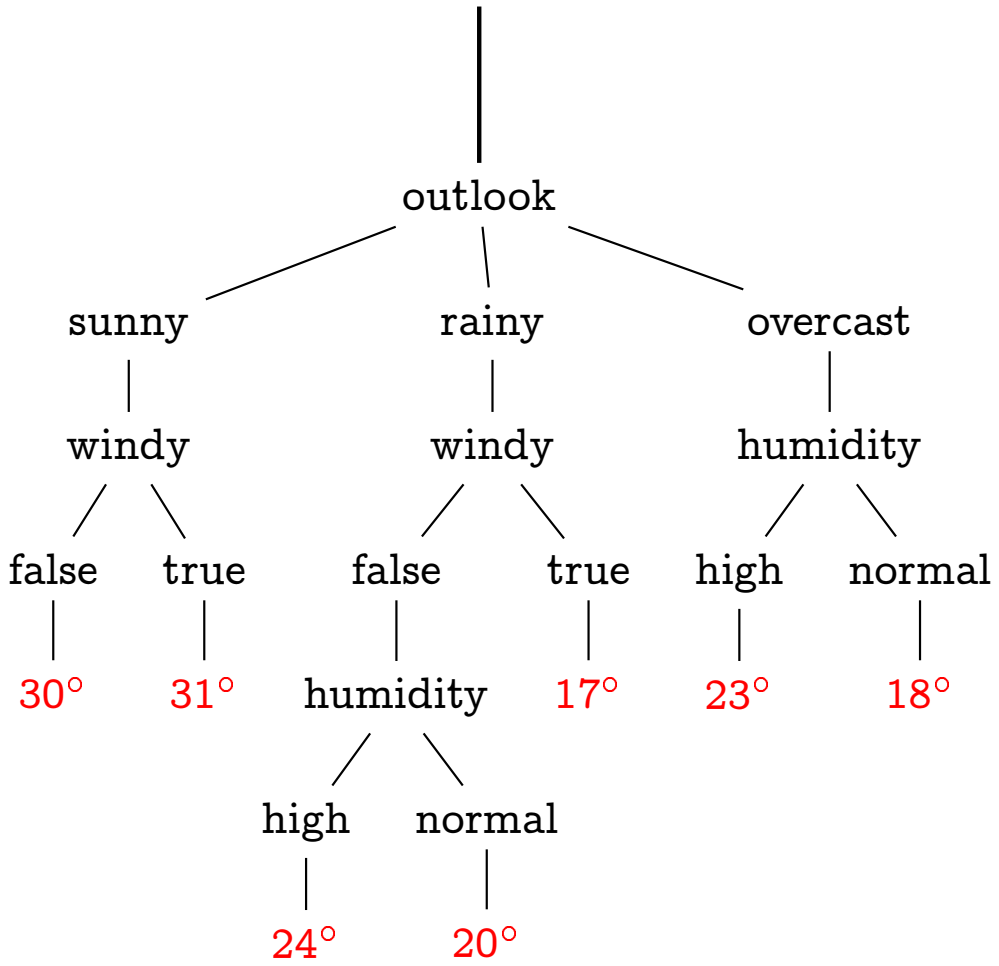


Same idea applies for regression

outlook	humidity	windy	temp.
sunny	high	false	30°
sunny	high	true	32°
overcast	high	false	23°
rainy	high	false	24°
rainy	normal	false	20°
rainy	normal	true	17°
overcast	normal	true	18°

Regression tree

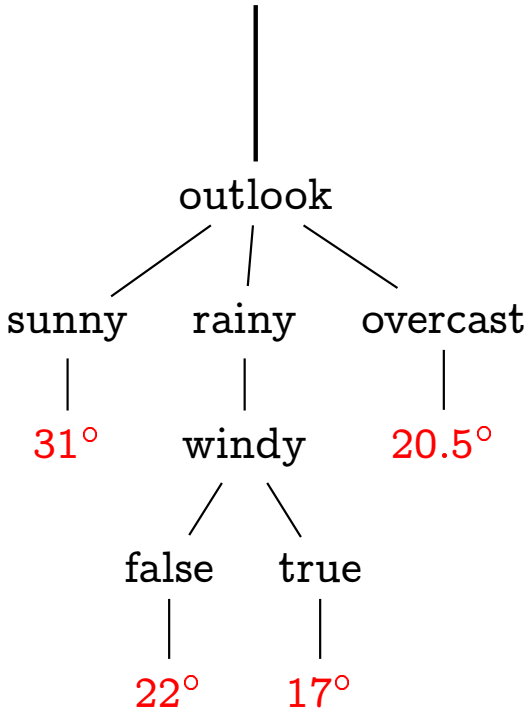
outlook	humidity	windy	temp.
sunny	high	false	30°
sunny	high	true	32°
overcast	high	false	23°
rainy	high	false	24°
rainy	normal	false	20°
rainy	normal	true	17°
overcast	normal	true	18°



Regression tree

We can also leave the tree not fully grown, and make the predictions by taking the average of output of every training observation at the given leaf in the tree.

outlook	humidity	windy	temp.
sunny	high	false	30°
sunny	high	true	32°
overcast	high	false	23°
rainy	high	false	24°
rainy	normal	false	20°
rainy	normal	true	17°
overcast	normal	true	18°

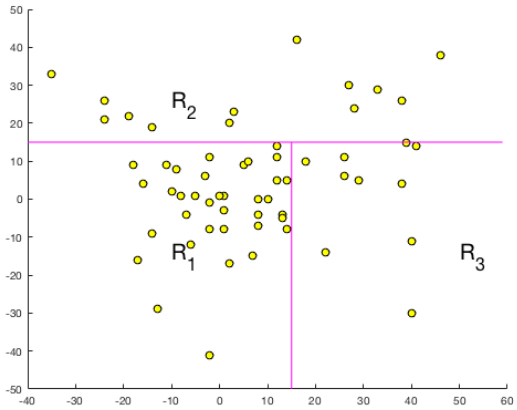


Main idea of building a (regression) decision tree

- Divide the feature space into J distinct non-overlapping regions, $R_1, \ldots, R_j$. Note that one region corresponds to one leaf in the tree.
- For every observation that falls into the region R_j, we make the same prediction, which is the mean of the responses of the observations in that region.

The essential problem is to find regions minimizing the MSE (*c.f.* cost function in regression)

$$\frac{1}{n} \sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2.$$



Clearly, in the case of non-contradicting data we can obtain zero training error.

How to part the feature space?

Finding all regions is computationally infeasible!

Solution: greedy top-down recursive search: *recursive binary split.*

- Greedy: locally optimal decisions
- Top-down: starts with only root node

Define by $R_1(j, s)$ and $R_2(j, s)$ the half-planes

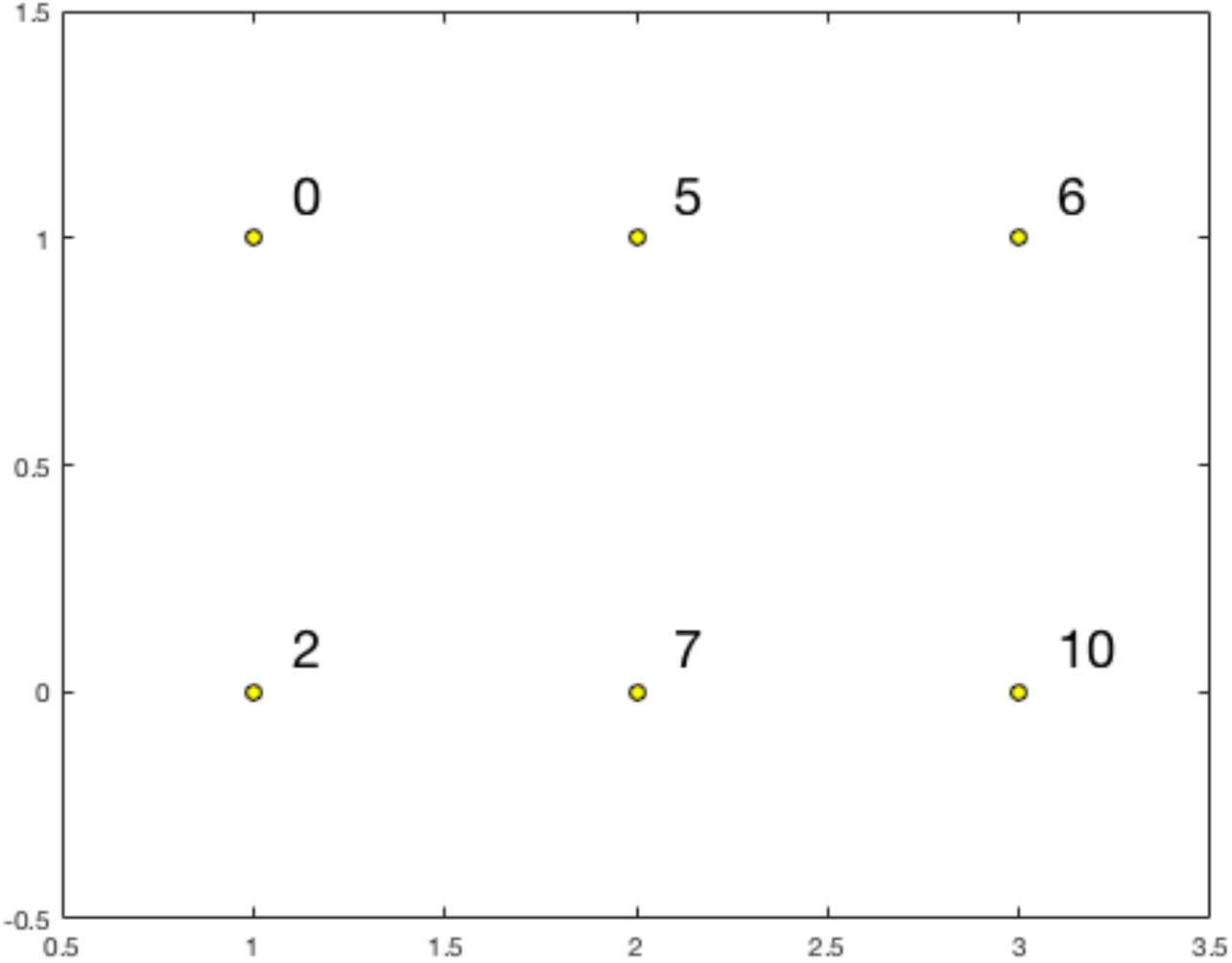
$$R_1(j, s) = \{x | x_j < s\}, \quad \text{and } R_2(j, s) = \{x | x_j \geq s\}.$$

We start by finding j, s such that the following expression is minimized

$$\sum_{i:x_i \in R_1(j,s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i:x_i \in R_2(j,s)} (y_i - \hat{y}_{R_2})^2.$$

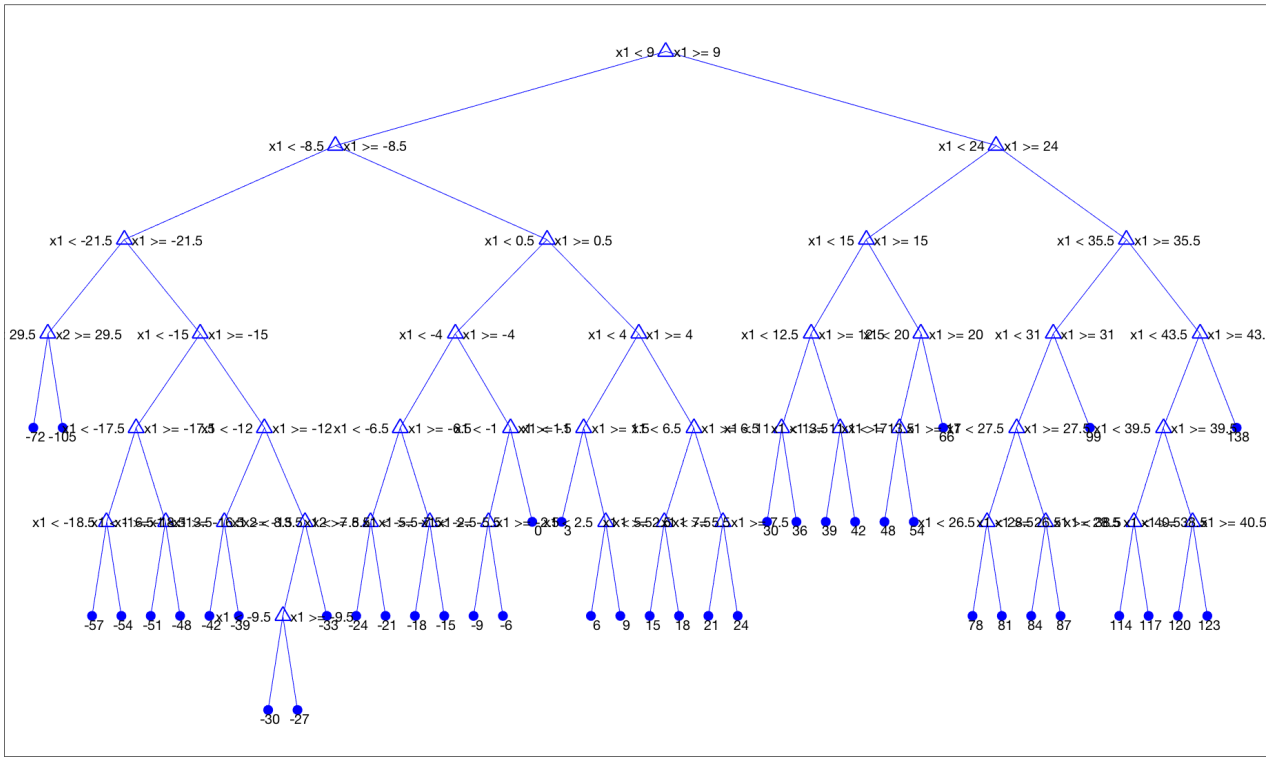
Once found, we apply the same procedure on the smaller half planes, until some stopping criteria is found. *E.g.* MATLAB stops when there are at most 10 examples in each leaf by default (or some other criteria for not being able to continue, like contradictive data).

Example



Trees are highly(!) prone to overfitting

Example of a tree where each instance is its own region, i.e. $J = n$



Avoid overfitting

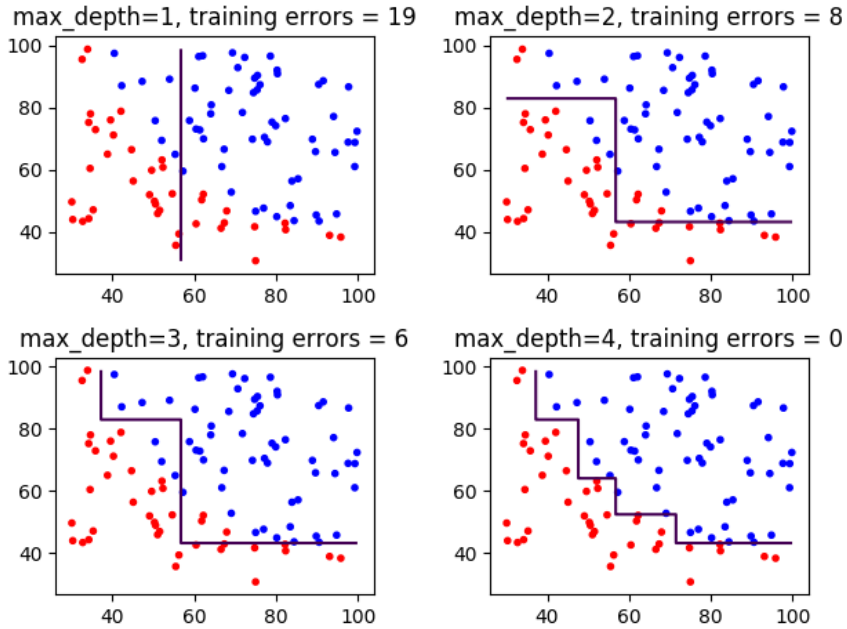
In order to avoid overfitting there are some tools that can be applied

- Control for the depth of the tree
- Control for the maximum number of splits
- Pruning...

Depth control

The main idea: grow a tree as deep as some predefined hyperparameter allows it to do.

There are several hyperparameters which could be set to prevent the tree from growing to deep. Two typical is the maximum depth of the tree and the maximum number of splits.



Pruning

The main idea: grow full tree and then prune in order to reduce the amount of 'unnecessary' branches.

As the number of subtrees grows exponentially it is (often) infeasible to test all of them.

One alternative is to use so-called *cost complexity pruning*. This method shares similarities with regularization as we've seen previously. We want to find a subtree T such that

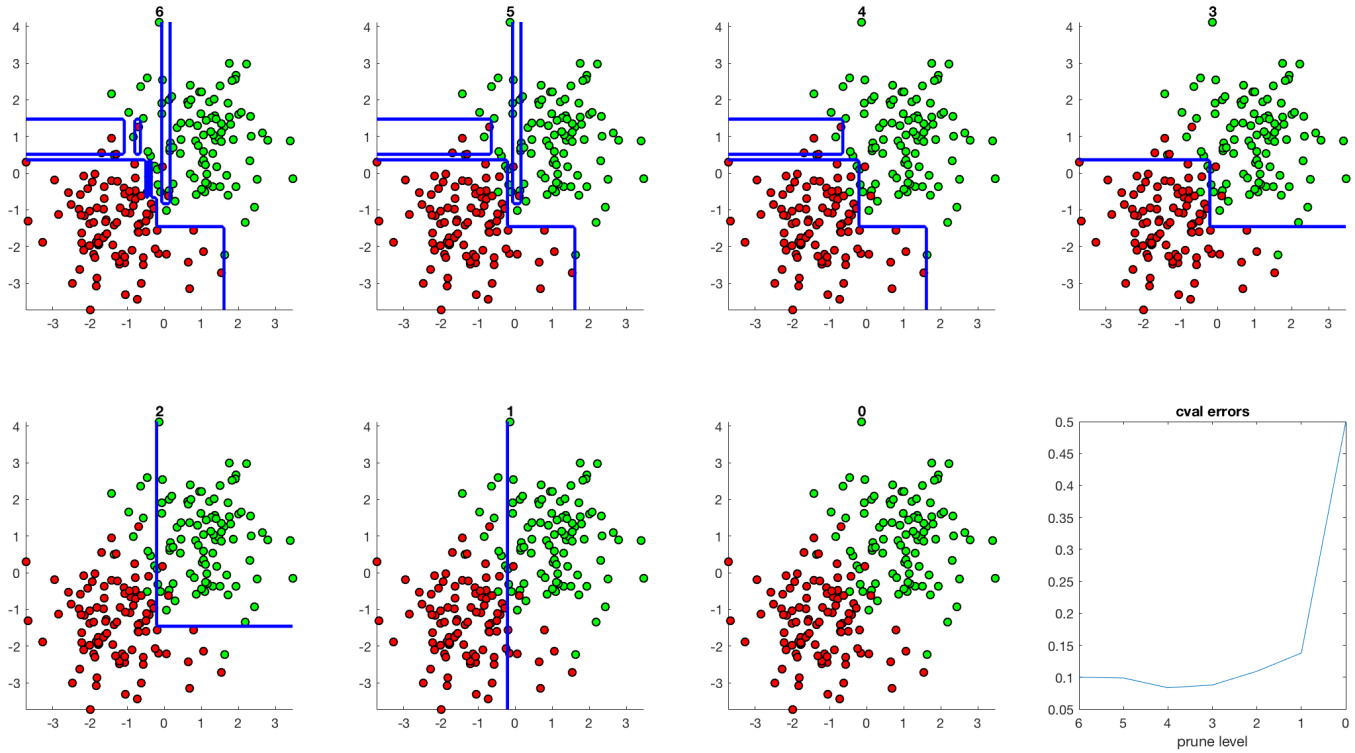
$$\sum_{m=1}^{|T|} \sum_{i:x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha |T|,$$

where $\alpha > 0$ is a tuning parameter and $|T|$ is the number of leaves of the tree.

- Small values for α yields less impact of the penalty, *i.e.* deeper tree
- Larger values for α rewards more shallow trees.

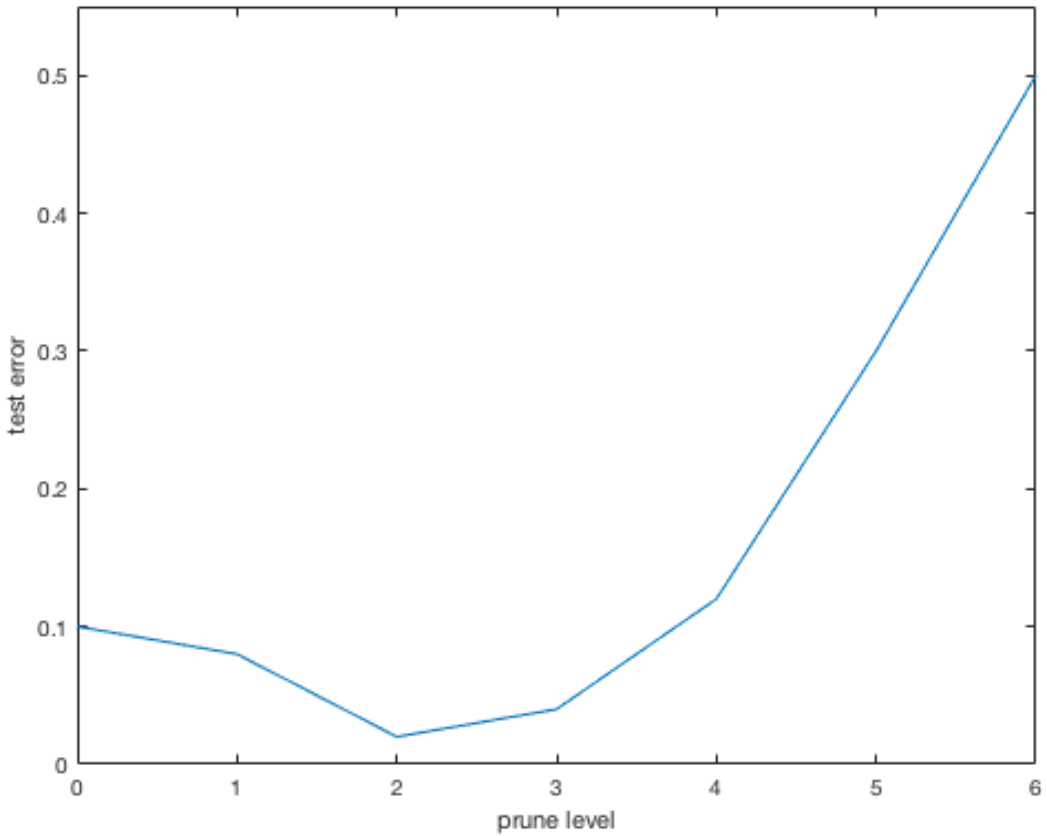
Optimal α can be found using a validation method.

Pruning example



Validation error versus α

Low prune level corresponds to small values for α, and prune level 0 is the unpruned tree. High prune level corresponds to large value for α and corresponds, and level 6 in this case is a tree without branches.



Classification setting

When we come to a leaf in the tree instead of using the mean of the instances as output we use mode, *i.e.* most frequently occurring class in said node (similar to what we did in kNN)

In the classification setting it does not make as much sense to use the MSE or RSS as a penalty to branch the tree. Instead we will consider

- Classification error
- Gini index
- Cross-entropy

What are the specific attributes of the decision boundary for decision trees?

Classification error

Assume that we have some classification problem with $K = 4$ classes A, B, C and D and want to use decision trees to model it

- Let T be a classification tree with J regions
- For each region let E_m denote the error of the mth region it is

$$E_m = 1 - \max_k(\hat{p}_{mk}),$$

  where $\hat{p}_{mk}$ is the proportion of instances in region m of the kth class.
- We define the cost of the tree as the cost of each individual region, i.e.

$$\sum_{m=1}^{J} E_m .$$

Example

In the mth leaf there are 10 instances distributed as follows: $6A, 2B, 1C, 1D$. Then $E_m = 1 - 0.6 = 0.4$.

Gini index

Assume that we have some classification problem with $K = 4$ classes A, B, C and D and want to use decision trees to model it

- Let T be a classification tree with J regions
- For each region let G_m denote the error of the mth region it is

$$G_m = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}) = 1 - \sum_{k=1}^{K} \hat{p}_{mk}^2.$$

- We define the cost of the tree as the cost of each individual region, i.e.

$$\sum_{m=1}^{J} G_m.$$

Example

In the mth leaf there are 10 instances distributed as follows: $6A, 2B, 1C, 1D$. Then

$$G_m = 0.6 \cdot 0.4 + 0.2 \cdot 0.8 + 0.1 \cdot 0.9 + 0.1 \cdot 0.9 = 0.58.$$

Cross-entropy

Assume that we have some classification problem with $K = 4$ classes A, B, C and D and want to use decision trees to model it

- Let T be a classification tree with J regions
- For each region let D_m denote the error of the mth region it is

$$D_m = -\sum_{k=1}^{K} \hat{p}_{mk} \log(\hat{p}_{mk}).$$

- We define the cost of the tree as the cost of each individual region, i.e.

$$\sum_{m=1}^{J} D_m.$$

Example

In the mth leaf there are 10 instances distributed as follows: $6A, 2B, 1C, 1D$. Then

$$D_m = -(0.6 \log(0.6) + 0.2 \log(0.2) + 0.1 \log(0.1) + 0.1 \log(0.1)) = 1.5710.$$

Comparing impurity measures

For the class distributions $6A, 2B, 1C, 1D$ we had

- $E_m = 1 - 0.6 = 0.4$
- $G_m = 0.6 \cdot 0.4 + 0.8 \cdot 0.2 + 2(0.1 \cdot 0.9) = 0.58$
- $D_m = \cdots = 1.5710$

For the class distribution $6A, 4B, 0C, 0D$ we obtain

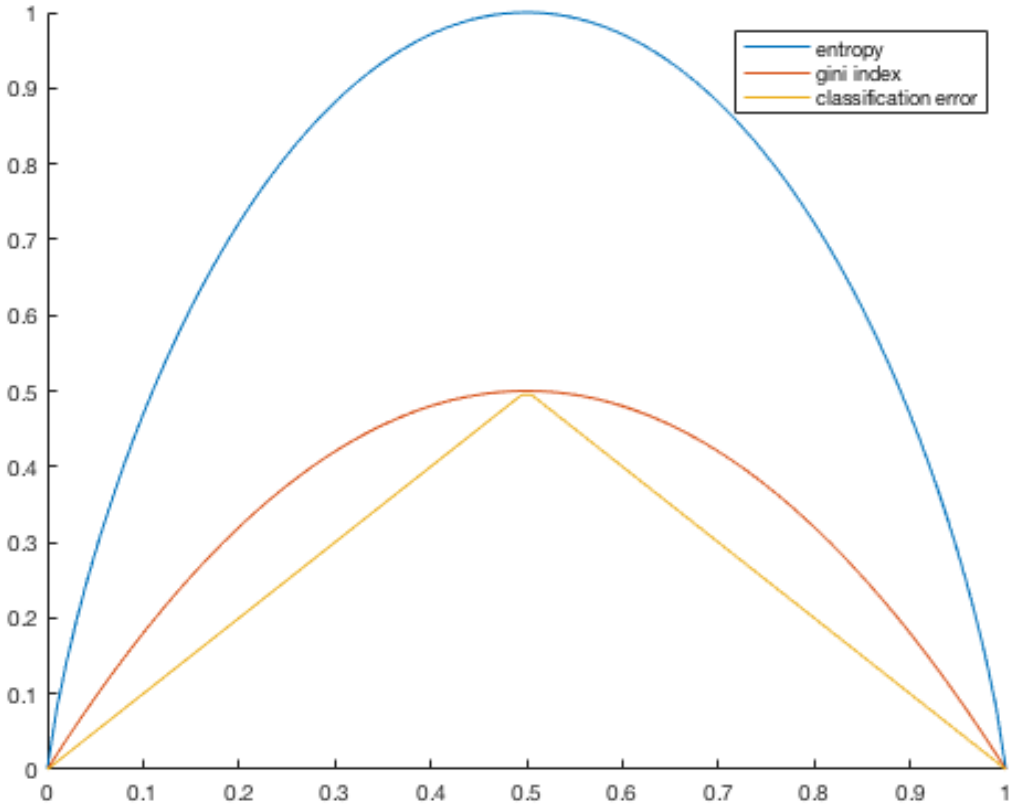
- $E_m = 1 - 0.6 = 0.4$
- $G_m = 2(0.6 \cdot 0.4) = 0.48$
- $D_m = -(0.6 \log(0.6) + 0.4 \log(0.4)) = 0.9710$

For the class distribution $10A, 0B, 0C, 0D$ we obtain

- $E_m = G_m = D_m = 0$

Conclusion: E_m is not as sensitive for node *purity* $\Rightarrow$ use it to measure performance not to grow the tree.

Comparing impurity measures



In general: $\max(\text{entropy}) = \log_2(n), \quad \max(\text{gini index}) = 1 - \frac{1}{n}.$

Purity gain Δ

Let v_j denote the elements in the jth leaf of a split at some part in the tree.

We define the purity gain Δ of that split as

$$\Delta = I(\text{parent}) - \sum_{j=1}^{k} \frac{N(v_j)}{N} I(v_j),$$

where I is our impurity measure either E, G or D, and $N(v_j)$ is the number of elements in the node v_j, and N is the number of elements in the parent node.

Pick the split with highest purity gain.

Example best split

Suppose we are at some part of the tree and the following data is what we have to use

humidity	windy	play
high	false	yes
high	true	no
high	false	yes
high	true	no
normal	false	yes
normal	true	no
normal	true	yes
normal	false	yes
normal	true	no
normal	true	yes

Gini index

$$I(\text{parent}) = 2(0.6 \cdot 0.4) = 0.48$$

Entropy

$$I(\text{parent}) = -\frac{4}{10} \cdot \log \frac{4}{10} - \frac{6}{10} \cdot \log \frac{6}{10} = 0.9710$$

Gini index

$$\Delta_{\text{hum}} = 0.48 - (0.4 \cdot 0.5 + 0.6 \cdot 4/9) = 0.0133$$

$$\Delta_{\text{wind}} = 0.48 - (0 + 0.6 \cdot 4/9) = 0.2133$$

Entropy

$$\Delta_{\text{hum}} = 0.9710 - (0.4 \cdot 1 + 0.6 \cdot 0.9183) = 0.02$$

$$\Delta_{\text{wind}} = 0.9710 - (0 + 0.6 \cdot 0.9183) = 0.42.$$

Advantages with decision trees

- Simple to understand and to interpret. Trees can be visualised.
- Can handle both categorical and numerical data, and thus often requires little data preparation
- Can to some extent handle missing data
- The cost of prediction is logarithmic in the number of data points used to train the tree $\Rightarrow$ cheaper than kNN, but more expensive than logistic regression
- Can handle multiclass problems without having to resort to other techniques

However...

- Non-robust, *i.e.* small change in data might yield large change in model
- Can easily overfit the data
- As it is NP-hard to find the optimal tree, using greedy methods only provides us with locally optimal improvements.

Decision trees in sklearn

In sklearn the command for creating a decision tree is given by
`from sklearn import tree`

- `tree.DecisionTreeClassifier()` for classification trees
- `tree.DecisionTreeRegressor()` for regression trees.

Pruning is currently *not* supported in sklearn.

However, there are plenty of name-value-pairs which can be passed as arguments in the construction of the tree, please see the documentation. Some important are

- `'max_depth' select the maximum depth of the tree`
- `'criterion'` select impurity measure for splitting branches `gini` for gini index and `entropy` for entropy.

Wisdom of the crowd

Imagine a biased coin which has a 51% to turn out as heads.

On a single toss, there is a obviously a 51% chance of turning out as heads.

What happens if we toss the coin n times? What is the probability that it is heads in *most* cases?

Wisdom of the crowd

Imagine a biased coin which has a 51% to turn out as heads.

On a single toss, there is a obviously a 51% chance of turning out as heads.

What happens if we toss the coin n times? What is the probability that it is heads in *most* cases?

The probability of heads in *exactly* k out of n tosses is given by

$$P(k \text{ heads}) = \binom{n}{k} p^k (1-p)^{n-k},$$

where p is the probability for heads.

Hence, if $n = 1,000$, then it will be heads most of the time if $k > 500$, and since these events are disjoint we may sum their corresponding probabilities, and we get

$$P(\text{most of the time heads}) = \sum_{k=501}^{1000} \binom{1000}{k} (.51)^k (.49)^{1000-k} \approx .73.$$

Wisdom of the crowd

Either by increasing the number of tosses, or finding an even more biased coin (to our advantage) we may increase the probability of obtaining heads in most cases.

For example if the probability of heads is 55%, then $n = 1,000$ would yield a 99.9% chance of turning out heads most of the time.

Tossing the original coin $n = 10,000$ times will yield a probability above 97%.

Wisdom of the crowd

Either by increasing the number of tosses, or finding an even more biased coin (to our advantage) we may increase the probability of obtaining heads in most cases.

For example if the probability of heads is 55%, then $n = 1,000$ would yield a 99.9% chance of turning out heads most of the time.

Tossing the original coin $n = 10,000$ times will yield a probability above 97%.

How is this related to machine learning?

Wisdom of the crowd

Either by increasing the number of tosses, or finding an even more biased coin (to our advantage) we may increase the probability of obtaining heads in most cases.

For example if the probability of heads is 55%, then $n = 1,000$ would yield a 99.9% chance of turning out heads most of the time.

Tossing the original coin $n = 10,000$ times will yield a probability above 97%.

How is this related to machine learning? Well, suppose that we have 1,000 (independent) poor classifiers each with a 51% chance (called *weak learners*) of classifying something correctly. Then a majority vote of these classifiers could yield a correct prediction in $\sim 75\%$ of all cases. This is what is known as the *wisdom of the crowd*.

Wisdom of the crowd

Either by increasing the number of tosses, or finding an even more biased coin (to our advantage) we may increase the probability of obtaining heads in most cases.

For example if the probability of heads is 55%, then $n = 1,000$ would yield a 99.9% chance of turning out heads most of the time.

Tossing the original coin $n = 10,000$ times will yield a probability above 97%.

How is this related to machine learning? Well, suppose that we have 1,000 (independent) poor classifiers each with a 51% chance (called *weak learners*) of classifying something correctly. Then a majority vote of these classifiers could yield a correct prediction in $\sim 75\%$ of all cases. This is what is known as the *wisdom of the crowd*.

Although, note that

- Your ML-models are typically *not* independent
- *But* your learners might also not be as weak as .51 accuracy.

Voting classifiers

Let k different models serve as your ensemble, train them separately, and when it comes to predictions of a new point x you simply let all classifiers cast a single vote (might be weighted) and then predication of x is that of the majority.

For classifiers with the capacity of predicting probabilities this can be used instead of the mode to vote. This is known as *soft voting* compared to *hard voting* when the mode is used.

Figure: Voting classifier here with 4 decision trees, but note that it may contain any type of classifiers

Bagging and pasting

Another approach to diversify the ensemble is to use the same training algorithms but trained on different subsets of the training data.

If this sampling of the training data is done *with* replacement then this approach is called *bagging* otherwise it is known as *pasting*.

Generally this yields a decrease in variance.

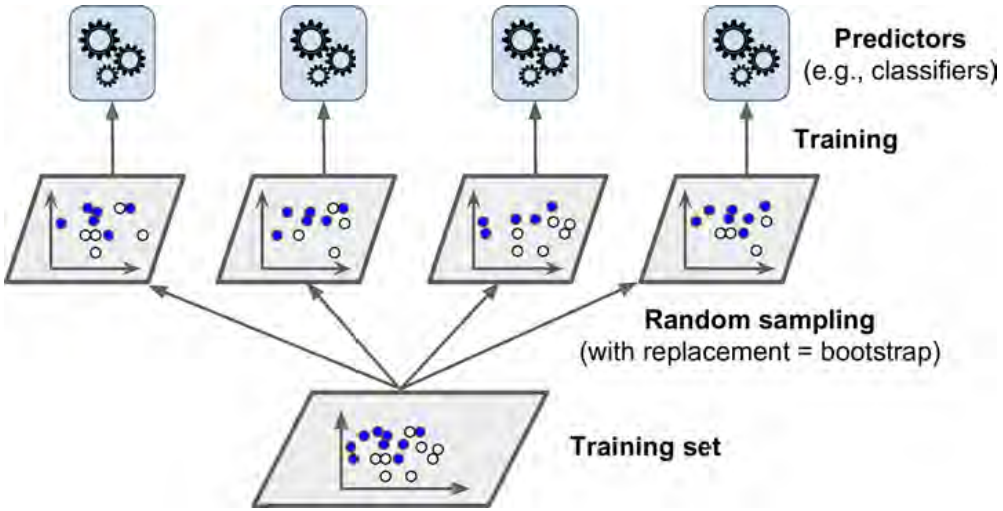


Figure: Illustration of bagging and pasting procedure

Bagging is variance reducing

Let $z_1, z_2, \ldots, z_B$ be identically distributed random variables, such that

$$\mathbb{E}[z_b] = \mu, \qquad \text{Var}[z_b] = \sigma^2,$$

and such that the average correlation is ρ. Then we get

$$\mathbb{E}\left[\frac{1}{B}\sum_{b=1}^{B} z_b\right] = \frac{1}{B}(\mu + \mu + \cdots + \mu) = \mu$$

and

$$\text{Var}\left[\frac{1}{B}\sum_{b=1}^{B} z_b\right] = \frac{1-\rho}{B}\sigma^2 + \rho\sigma^2.$$

Out-of-bag evaluation

Assume that we have a regular six-sided dice, and we throw this 6 times. Is it likely that every side will turn up?

Out-of-bag evaluation

Assume that we have a regular six-sided dice, and we throw this 6 times. Is it likely that every side will turn up? No, the probability that the six outcomes are $1, 2, 3, 4, 5$ and 6 is

$$P(\text{every side once}) = 1 \cdot \frac{5}{6} \cdot \frac{4}{6} \cdot \frac{3}{6} \cdot \frac{2}{6} \cdot \frac{1}{6} = \frac{5}{324}.$$

Out-of-bag evaluation

Assume that we have a regular six-sided dice, and we throw this 6 times. Is it likely that every side will turn up? No, the probability that the six outcomes are $1, 2, 3, 4, 5$ and 6 is

$$P(\text{every side once}) = 1 \cdot \frac{5}{6} \cdot \frac{4}{6} \cdot \frac{3}{6} \cdot \frac{2}{6} \cdot \frac{1}{6} = \frac{5}{324}.$$

Hence, if the dice had 100 sides and we threw it 100 times the probability of every side coming up once is next to nothing, since

$$P(\text{every side once}) = \prod_{j=1}^{100} \frac{j}{100} = \frac{100!}{100^{100}} \approx 9.3 \cdot 10^{-43}.$$

Out-of-bag evaluation

Assume that we have a regular six-sided dice, and we throw this 6 times. Is it likely that every side will turn up? No, the probability that the six outcomes are $1, 2, 3, 4, 5$ and 6 is

$$P(\text{every side once}) = 1 \cdot \frac{5}{6} \cdot \frac{4}{6} \cdot \frac{3}{6} \cdot \frac{2}{6} \cdot \frac{1}{6} = \frac{5}{324}.$$

Hence, if the dice had 100 sides and we threw it 100 times the probability of every side coming up once is next to nothing, since

$$P(\text{every side once}) = \prod_{j=1}^{100} \frac{j}{100} = \frac{100!}{100^{100}} \approx 9.3 \cdot 10^{-43}.$$

So, how many sides are typically shown for 100 sided dices and 100 throws?

Out-of-bag evaluation

Assume that we have a regular six-sided dice, and we throw this 6 times. Is it likely that every side will turn up? No, the probability that the six outcomes are $1, 2, 3, 4, 5$ and 6 is

$$P(\text{every side once}) = 1 \cdot \frac{5}{6} \cdot \frac{4}{6} \cdot \frac{3}{6} \cdot \frac{2}{6} \cdot \frac{1}{6} = \frac{5}{324}.$$

Hence, if the dice had 100 sides and we threw it 100 times the probability of every side coming up once is next to nothing, since

$$P(\text{every side once}) = \prod_{j=1}^{100} \frac{j}{100} = \frac{100!}{100^{100}} \approx 9.3 \cdot 10^{-43}.$$

So, how many sides are typically shown for 100 sided dices and 100 throws? The probability of not being shown is $(1 - 1/n)$, so for n throws this probability is

$$\left(1 - \frac{1}{n}\right)^n \text{ and } \lim_{n \to \infty} \left(1 - \frac{1}{n}\right)^n = 1/e \approx 37\%.$$

Thus, only 63% of the sides are seen. Hence, in the context of bagging only 63% of the samples are seen by the predictor and the rest can be used for validation, without having to use a separate validation set.

Random forests

In short

$$\text{Decision trees} + \text{Bagging} + \text{Some randomness} = \text{Random Forests}.$$

- Decision trees are particularly good to use bagging ensembles on.
- Random forests was a solution proposed to solve the problem of decision trees overfitting. To utilize the strong properties but to work around the variance issues.
- The name forest is obvious as it grows several decision trees. These are then controlled by the use of bagging, the emsemble approach and some randomized limitations on what features can be used for each tree.
- In `sklearn` many of the hyperparameters for decision trees also applies to Random forests.
- Random forests has built-in feature importance identification, where you can look at how much a given feature reduce impurity on average over all trees. The feature which reduces impurity the most is then the most important feature. The importance of a features can be found in `sklearn` by `feature_importances_`.

Boosting

While bagging primary is a method for variance reduction, is boosting a method for bias boosting.

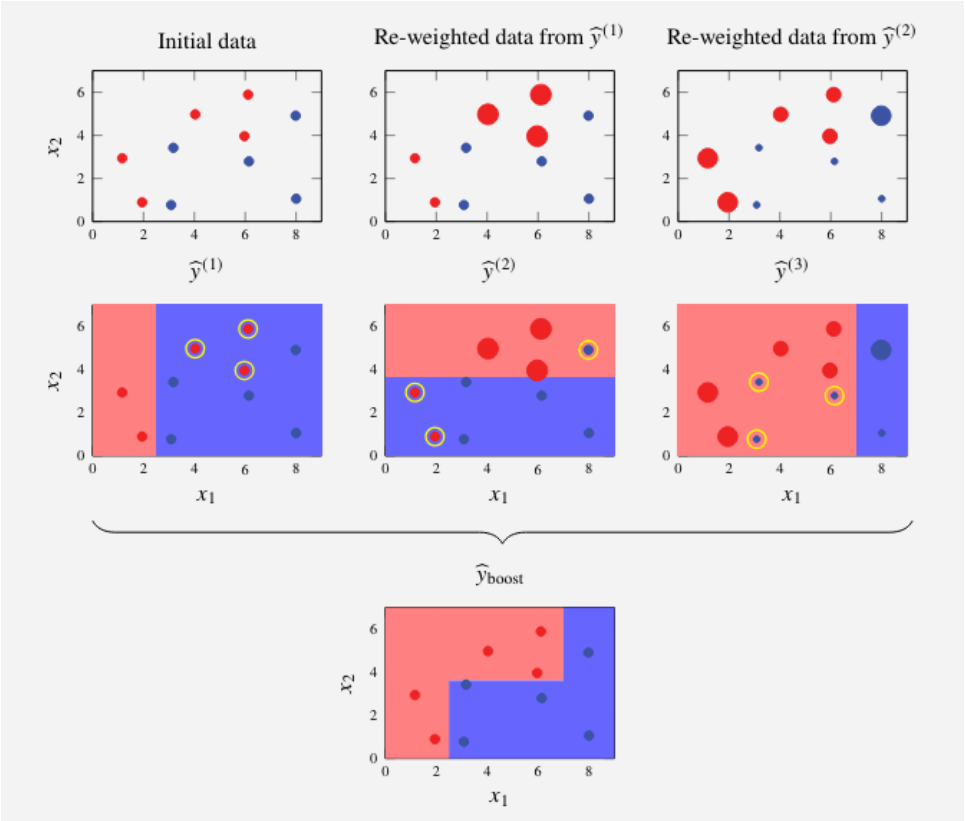


Figure 7.7 in Course book (Machine learning)

Assignment summary

This lecture comes with one exercise in Assignment 3.

1. Ensemble of Batman Trees: In this exercise you will your own simplified version of a random forest by means of using 100 decision trees and bootstraping. We will revisit the Batman dataset from the SVM exercise, which is a difficult task.

   Even though it is not mandatory for the assignment please compare your own "random forest" to the built-in version in sklearn.

Finding a good model

Finding a good model involves among other things

- Feature selection
- Hyperparamter tuning

We recall that hyperparameters are parameters which are set before the learning process begins.

Example of hyperparameters for decision trees are

- split criterion
- `max_depth`
- `min_samples_leaf`

Grid search is a typical approach for trying to find an optimal set of hyper-parameters