Decision tree learning

COMS 4771 Fall 2023

Decision trees

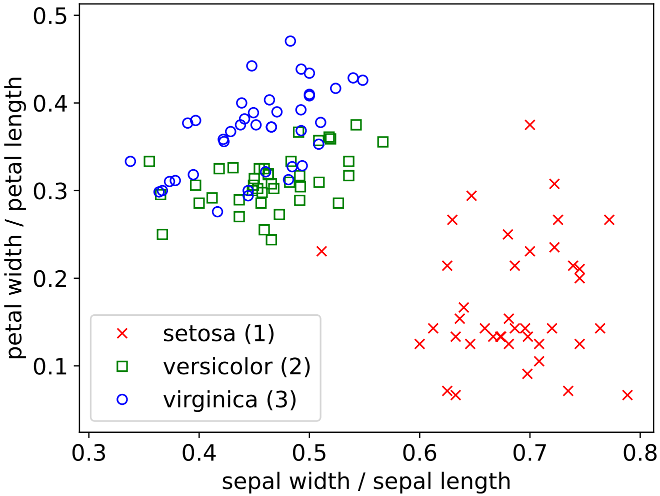
Decision trees: nested if-then-else statements

- Can be relatively easy to understand (when not too large)
- Can have fast execution time (when not too large)
- Standard learning algorithm has some nice properties

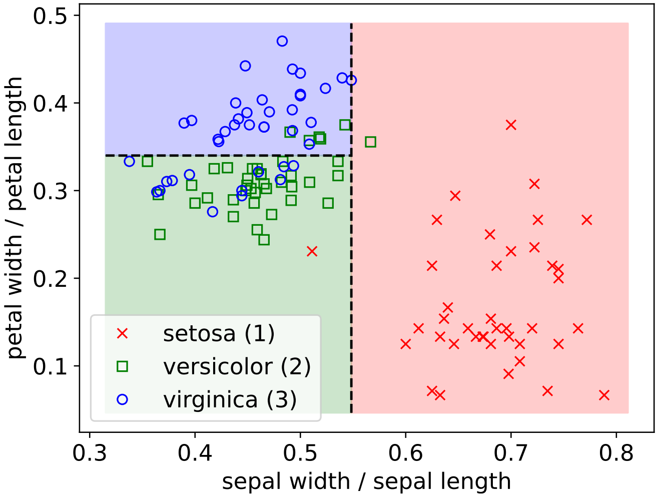
Decision trees vs nearest neighbors

- Both: try to exploit “local regularity”
- Nearest neighbors: memorize training data
- Decision trees: use training data to carve $\mathcal{X}$ into regions
  - . . . so that, for each region, there is a good constant prediction

Example: iris dataset (using different features)

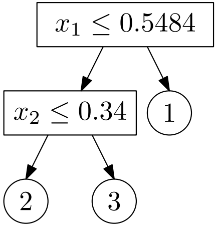


Example: iris dataset (using different features)



Structure of decision tree (in context of prediction):

- Rooted binary tree T
- A non-leaf node is associated with a predicate involving single feature
- A leaf node is associated with a label from $\mathcal{Y}$
- Computing $f_T(x) =$ prediction of tree T at x:
  Start at root node
  - If current node is leaf node: return associated label
  - Else if predicate at x is true: recurse on left child
  - Else: recurse on right child



Top-down learning algorithm

Top-down learning algorithm: repeatedly modify tree to reduce its "cost"

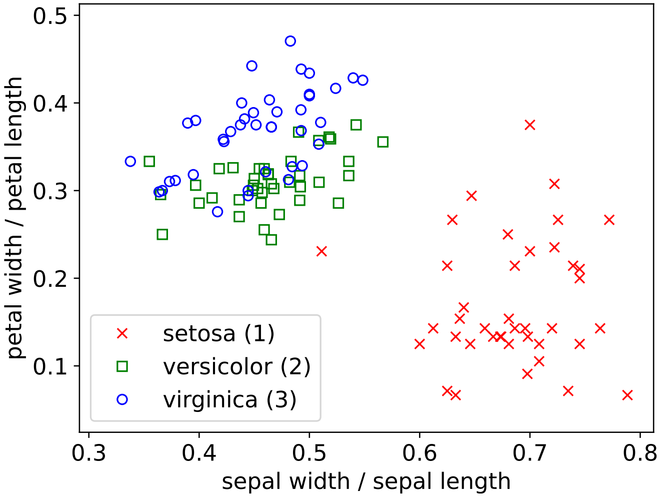
- Simplest cost function (for classification): training error rate

$$\widehat{\mathrm{err}}[f_T; \mathcal{S}] = \frac{1}{|\mathcal{S}|} \sum_{(x,y)\in\mathcal{S}} \mathbb{1}\{f_T(x) \neq y\}$$

- (Classification tree = decision tree for classification problem)

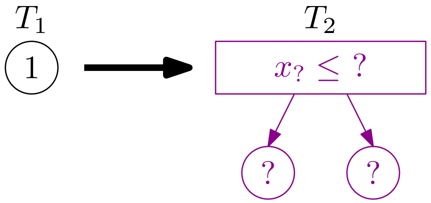
- Initial tree: a single (leaf) node
- Repeat until done: make a modification to tree that reduces the cost the most

Example: iris dataset (using different features)



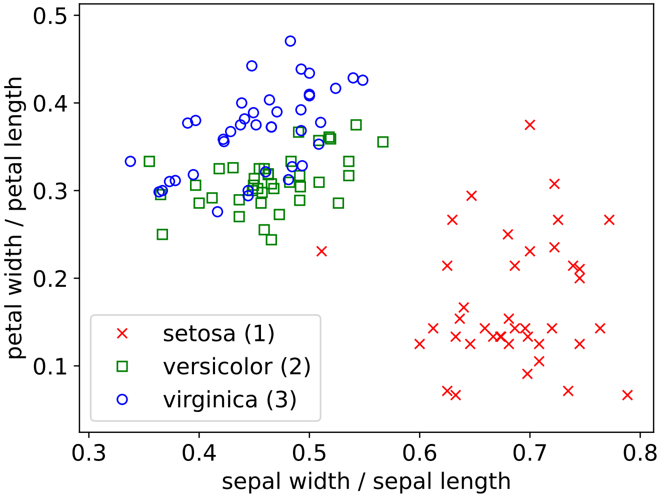
Allowed modifications to improve the tree:

- Replace a leaf node with a decision stump

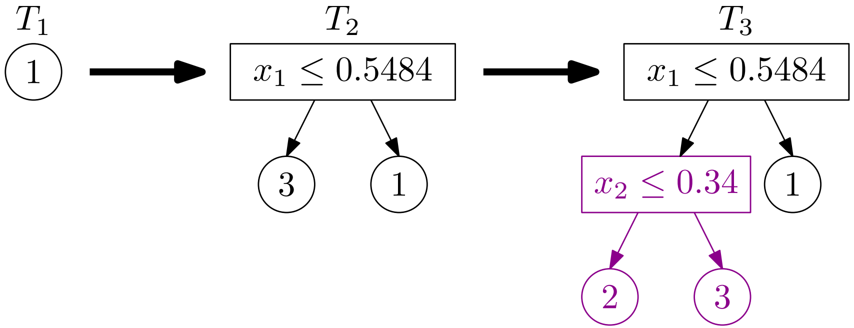


- How many possible modifications are there?

Example: iris dataset (using different features)



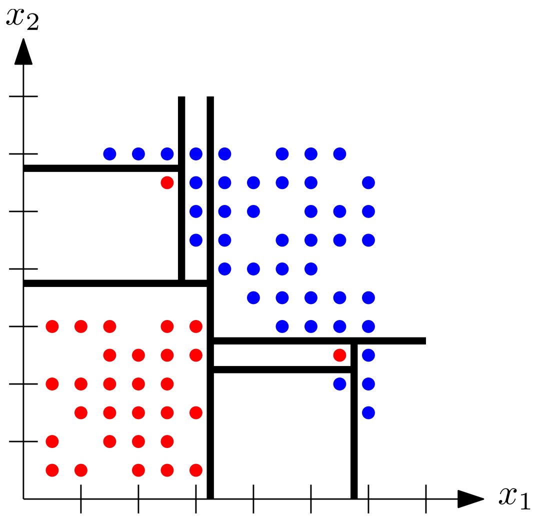
Two steps of top-down algorithm on iris dataset



When to stop modifying the tree? Some options:

- Stop when no modification leads to reduction in cost
- Stop when # leaves or depth reaches predetermined maximum
- Stop when each leaf node is "pure" (i.e., all training examples that "reach" the leaf node have same label or same feature vector)

Over-fitting training data

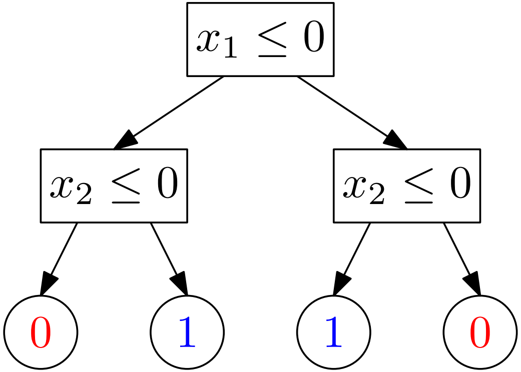


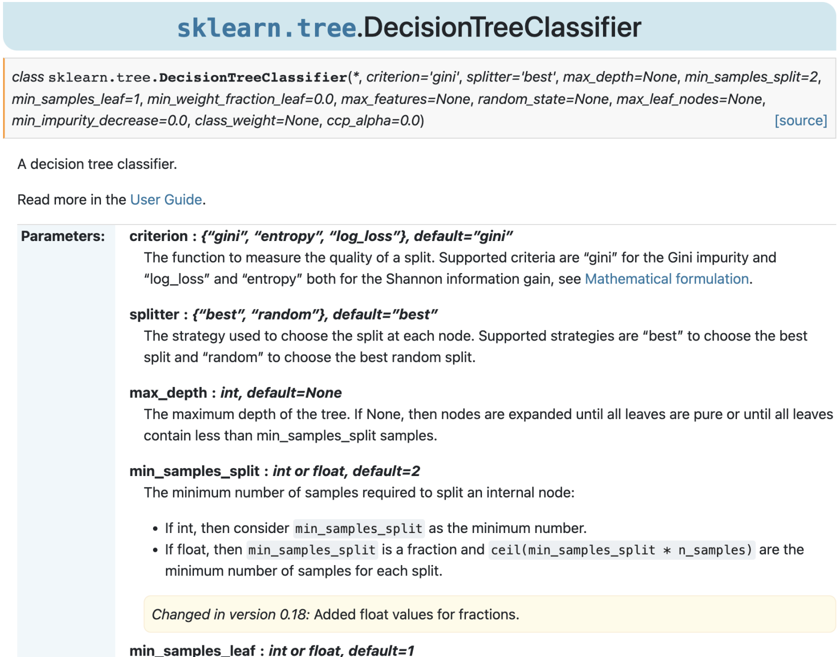
Training data

feature vector	label
$(0,0)$	0
$(0,1)$	1
$(1,0)$	1
$(1,1)$	0

Decision trees with 1 or 2 leaf nodes make 2 mistakes
(Myopic learner does not get past first step)

But the following makes no mistakes:





sklearn.tree.DecisionTreeClassifier

class `sklearn.tree.DecisionTreeClassifier`(**, criterion='gini', splitter='best', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None, random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0, class_weight=None, ccp_alpha=0.0*) [source]

A decision tree classifier.

Read more in the User Guide.

Parameters:

criterion : *{"gini", "entropy", "log_loss"}, default="gini"*

The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "log_loss" and "entropy" both for the Shannon information gain, see Mathematical formulation.

splitter : *{"best", "random"}, default="best"*

The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.

max_depth : *int, default=None*

The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split : *int or float, default=2*

The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf : *int or float, default=1*

Regression trees

Regression trees: decision trees for real-valued prediction, (usually) with squared error as loss function

- Q: How to determine the labels associated with the leaf nodes?
- A: Average of labels among examples that "reach" the leaf node

Model averaging

Suppose you have many possible predictors $f_1, f_2, \ldots, f_T$
(or many possible ways to learn a predictor)

- Model selection: try to choose the best one
- Model averaging: combine them into a single predictor by averaging/voting

Simplest form: uniform model averaging

$$f_{\text{avg}}(x) = \frac{1}{T} \sum_{t=1}^{T} f_t(x)$$

(For classification, use majority/plurality vote instead of averaging)

$$\underbrace{\mathbb{E}\big[(f_{\text{avg}}(X) - Y)^2\big]}_{\text{mse}[f_{\text{avg}}]} = \frac{1}{T}\sum_{t=1}^{T}\underbrace{\mathbb{E}\big[(f_t(X) - Y)^2\big]}_{\text{mse}[f_t]} \\ \underbrace{- \frac{1}{2T^2}\sum_{s=1}^{T}\sum_{t=1}^{T}\mathbb{E}\big[(f_s(X) - f_t(X))^2\big]}_{\text{average disagreement}}$$

To generate many “similar” predcitors that may disagree often:

- Train each predictor on a different (random) subset of the training data

Popular alternative: Bootstrap resampling of $\mathcal{S} = ((x^{(i)}, y^{(i)}))_{i=1}^{n}$

- Independently sample T new datasets $\mathcal{S}^{(1)}, \ldots, \mathcal{S}^{(T)}$, where

$$\mathcal{S}^{(t)} = ((X^{(t,i)}, Y^{(t,i)}))_{i=1}^{n} \overset{\text{i.i.d.}}{\sim} \mathrm{Unif}(\mathcal{S})$$

  - Differs from “sampling without replacement”
  - Some examples in $\mathcal{S}$ can appear more than once in $\mathcal{S}^{(t)}$
  - Some may not appear at all

Bagging = bootstrap resampling + model averaging

- Use bootstrap resampling to generate $\mathcal{S}^{(1)}, \dots, \mathcal{S}^{(T)}$
- For each $t = 1, \dots, T$:
  Let $f_t =$ output of learning algorithm on $\mathcal{S}^{(t)}$
- Combine $f_1, \dots, f_T$ to form f_{avg} using uniform model averaging
  (Or f_{vote} using plurality vote, in case of classification problems)

```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.utils import resample
from scipy.stats import mode

def learn(train_x, train_y, num_trees=20):
  return [DecisionTreeClassifier().fit(*resample(train_x, train_y))
   ↪  for i in range(num_trees)]

def predict(params, test_x):
  predictions = np.array([tree.predict(test_x) for tree in params])
  return mode(predictions, axis=0, keepdims=False)[0]
```

Forest cover type dataset[1]

Problem: Create a program that, given cartographic data about a 30×30 meter region of a forest, predict the type of forest cover

- Dataset: "[...] four wilderness areas located in the Roosevelt National Forest of northern Colorado [...] minimal human-caused disturbances [...] forest cover types are more a result of ecological processes rather than forest management practices."
- Classes: spruce/fir (1), lodgepole pine (2), . . . , krummholz (7)
- Features ($d = 54$): elevation, slope, . . . , distance to water, distance to roads, . . . , amount of shade at 9am, amount of shade at 12pm, . . .
- Number of training data: 464809; number of test data: 116203

[1] `https://archive.ics.uci.edu/dataset/31/covertype`

Results on cover type

- Decision tree with trained by top-down algorithm
  - Stopped when all leaf nodes are pure
  - Test error rate: 6.1%
- Bagging + top-down as before ($T = 20$)
  - Individual trees' test error rates: between 7.7% and 8.0%
  - Plurality vote classifier test error rate: 3.5%