

Decision trees and how to prune them

Notes on Andrew Ng course

29 of October, 2022

Contents

Decision trees.....	3
Measuring purity: Entropy function.....	3
Choosing a split: Information gain.....	4
The algorithm.....	5
Continuous features.....	5
Regression trees.....	6
Tree ensembles.....	7
Using multiple decision trees.....	7
Building a forest: sampling with replacement.....	7
Random forest algorithm.....	8
XGBoost.....	9
When to use decision trees.....	10

Decision trees

Decision trees have as components:

- Decision nodes: where the algorithm decides which branch to follow based on a condition.
- Branches: the route the algorithm takes to the next node.
- Leaf nodes: final nodes and where the algorithm makes the decision.
- Root node: the first node of the algorithm.

For the learning process, there are 2 key points:

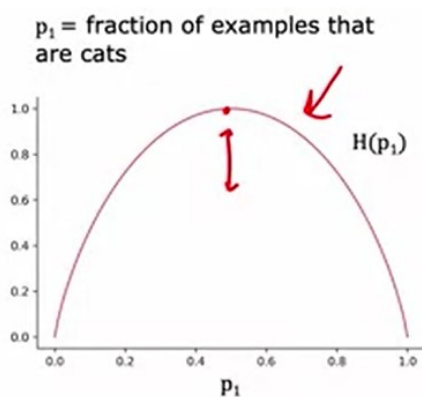
1. How to choose which feature to split in each node? We want to maximize purity
2. When do you stop splitting? 4 criteria:
 - i. When a node is 100% one class.
 - ii. When splitting a node results in the tree exceeding a maximum depth we choose (by keeping a tree small it makes it less prone to overfitting).
 - iii. When improvements in purity are below a certain threshold.
 - iv. When the number of examples in a node is below a threshold (again, to prevent overfitting).

Measuring purity: Entropy function

Entropy is the measure of the level of impurity. It takes the value:

- 0 when all examples are from the same class.
- 1 when examples are split 50/50.

The entropy function is:



$$p_0 = 1 - p_1$$

$$\begin{aligned} H(p_1) &= -p_1 \log_2(p_1) - p_0 \log_2(p_0) \\ &= -p_1 \log_2(p_1) - (1 - p_1) \log_2(1 - p_1) \end{aligned}$$

Note: " $0 \log(0)$ " = 0

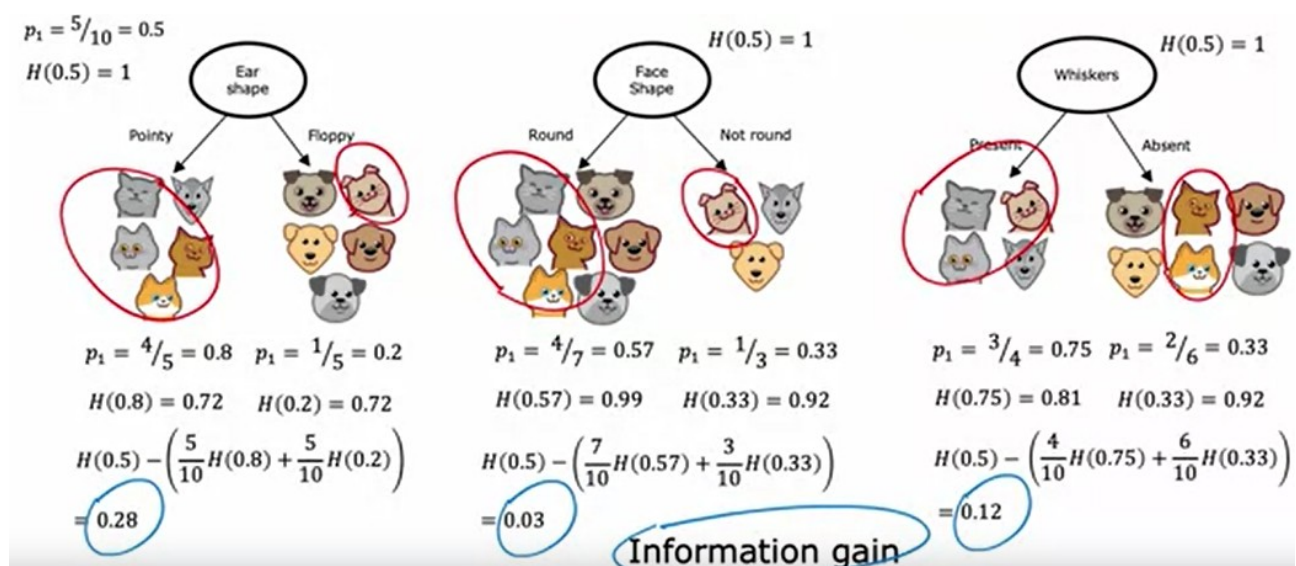
Which looks very similar to the logloss function for logistic regression (and there is a mathematical explanation behind this, which Andrew does not go into). Also it uses base 2 log in order to have 1 as its maximum, so results are easier to interpret.

There are other criteria to measure impurity, but we will focus in the Entropy function.

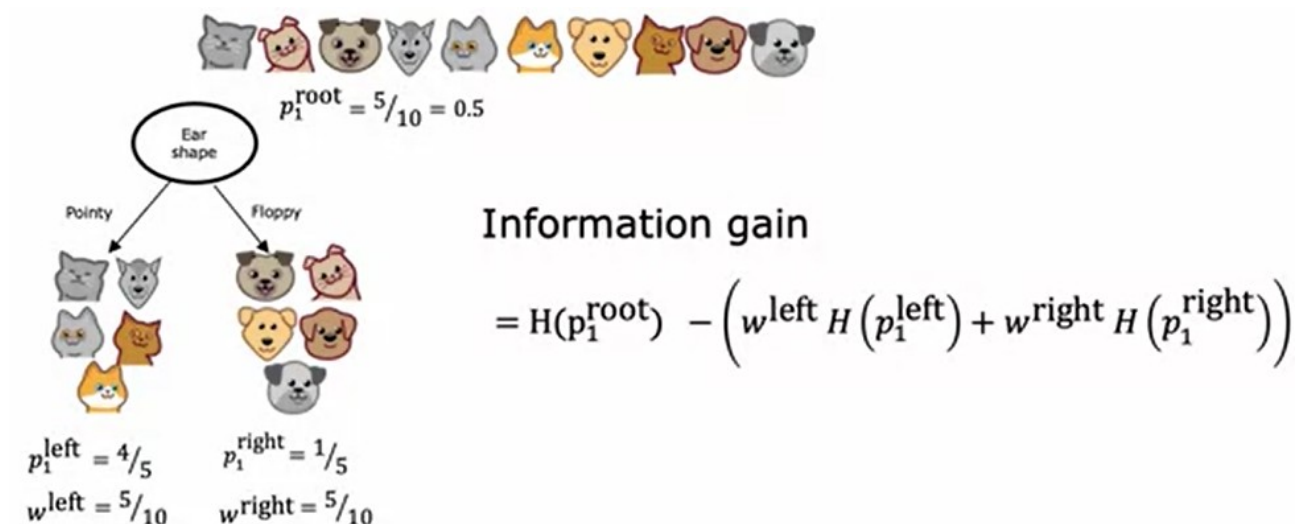
Choosing a split: Information gain

We will decide based on which feature reduces entropy the most, or reduces impurity or maximizes purity (they all mean the same). In decision tree learning the reduction of entropy is called information gain.

To calculate information gain, we need to consider the entropy and weight for each branch (because we want to give value to how many examples went to each branch, not only its entropy) and also the entropy of the previous node:



We choose the value with the highest information gain. The formal definition is:










The algorithm

This is the learning process of the recursive algorithm for decision trees:

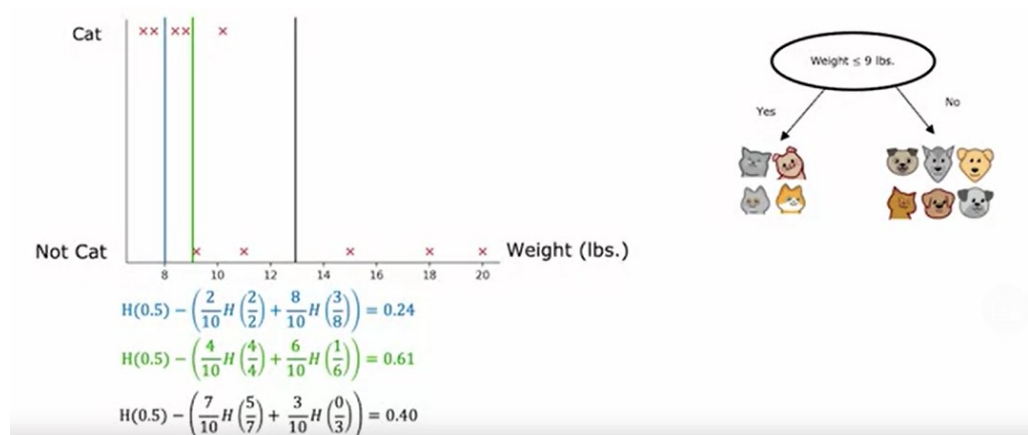
1. Start with all example at the root node.
2. Calculate information gain for all possible features, and pick the one with the highest information gain.
3. Split dataset according to selected feature, and create left and right branches of the tree.
4. Keep repeating the splitting process until stopping criteria is met:
 - i. When a node is 100% one class.
 - ii. When splitting a node will result in the tree exceeding a maximum depth.
 - iii. Information gain from additional splits is less than threshold.
 - iv. When number of examples in a node is below a threshold.

Continuous features

We can also split a node based on a continuous feature. Looking at the example to predict if an animal is a cat, we now include the weight as a feature:







	Ear shape	Face shape	Whiskers	Weight (lbs.)	Cat
	Pointy	Round	Present	7.2	1
	Floppy	Not round	Present	8.8	1
	Floppy	Round	Absent	15	0
	Pointy	Not round	Present	9.2	0
	Pointy	Round	Present	8.4	1
	Pointy	Round	Absent	7.6	1
	Floppy	Not round	Absent	11	0

When choosing the root node, just as before we test all features to find the one with more information gain. When testing the continuous feature (the weight), we try different values and see how it splits the target, calculating the information gain just as before. In the cat case that follows, we try for weights ≤ 8 , ≤ 9 , ≤ 13 , and we see that for weight ≤ 9 the information gain is the highest. If there was no feature with higher gain, we would choose this one.

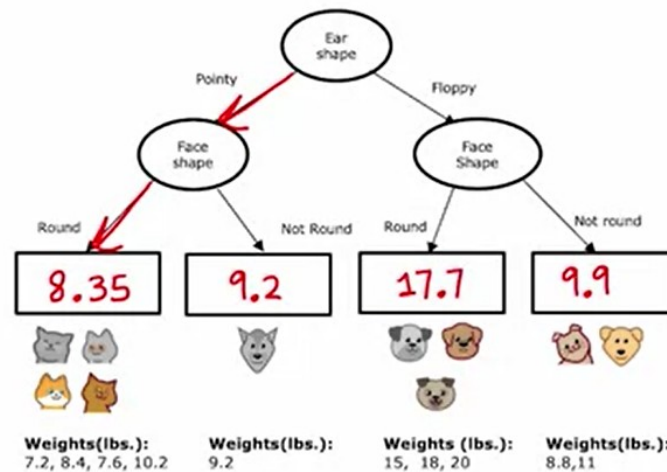


Regression trees

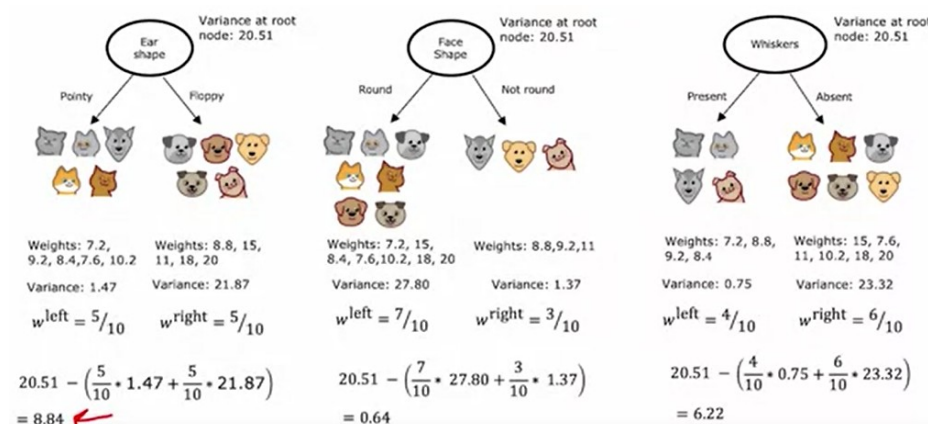
We can also use decision trees for regression, like the one below - this time we will consider the Weight as the target feature:

	Ear shape	Face shape	Whiskers	Weight (lbs.)
	Pointy	Round	Present	7.2
	Floppy	Not round	Present	8.8
	Floppy	Round	Absent	15
	Pointy	Not round	Present	9.2
	Pointy	Round	Present	8.4
	Pointy	Round	Absent	7.6

Be aware that, in the example below, the level one subtrees choose the same feature - there is no problem with that. For regression trees, the algorithm will predict the average of the examples of each leaf:



For choosing the split in a regression tree, instead of trying to reduce impurity or entropy, **we want to choose the option that gives the largest reduction in variance**. So we do the weighted average of the variations of each branch and subtract it from the previous' node variance, choosing the one with the highest variance reduction (in this case, the first one):

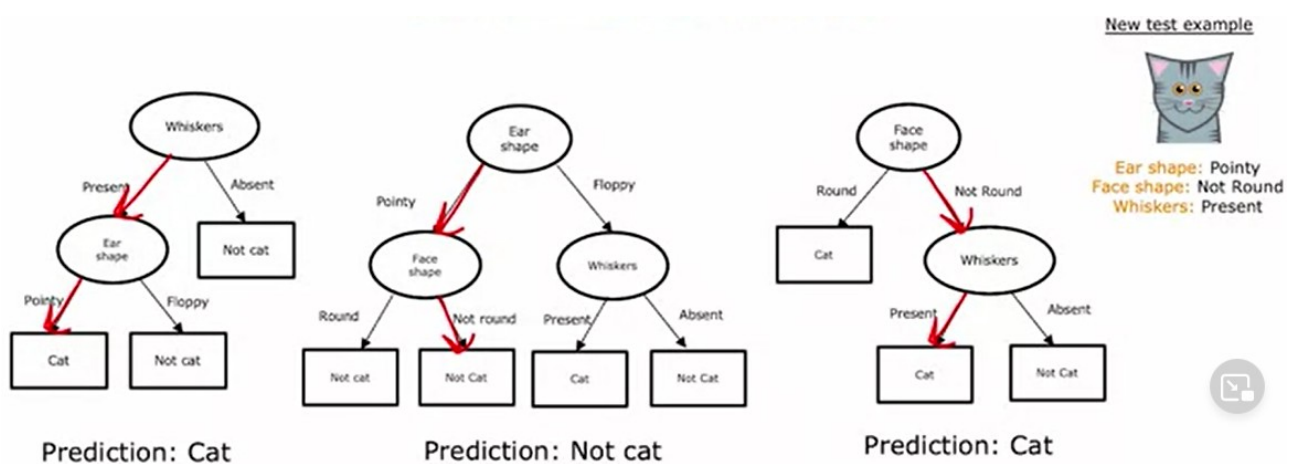


Tree ensembles

Using multiple decision trees

One of the weaknesses of using a single decision tree is that it can be sensitive to small changes in the data - small changes can produce completely different trees. One solution to make the algorithm more robust is to build many decision trees: that is called a tree ensemble.


We build different trees (in the next section we explain how) and then we run an example: the result obtained by the majority is the one we take. In the example below, 2 trees out of 3 predict a cat, so we predict "Cat":













Building a forest: sampling with replacement

Sampling with replacement is when, in the context of probability, we take a card from a deck, we put it back, and then we pick another one (so they would be independent random events). Sampling without replacement would be to take a card, put it aside, then take another one, etc (so they would be dependent events).

For building subsets in the decision tree, if we have a training set of 10 examples, we would create a subset by choosing 10 random examples with replacement from the training set (this means, any examples can be repeated up to 10 times). For example:



	Ear shape	Face shape	Whiskers	Cat
	Pointy	Round	Present	1
	Floppy	Not round	Absent	0
	Pointy	Round	Absent	1
	Pointy	Not round	Present	0
	Floppy	Not round	Absent	0
	Pointy	Round	Absent	1
	Pointy	Round	Present	1
	Floppy	Not round	Present	1
	Floppy	Round	Absent	0
	Pointy	Round	Absent	1

In this case, we can see there are several examples that repeat themselves, and that's ok!

This way, we create a training set that is quite different to the original one, but also pretty similar - what we need for planting a ensemble.

Random forest algorithm

We will build a random forest, which is a type of forest ensemble that works much better than a decision tree.

First we will build what is called a *bagged tree*: it is called like this because the sampling with replacement is sometimes compared with picking samples from a bag (where we can't see the content) and then putting them back before picking up new samples.

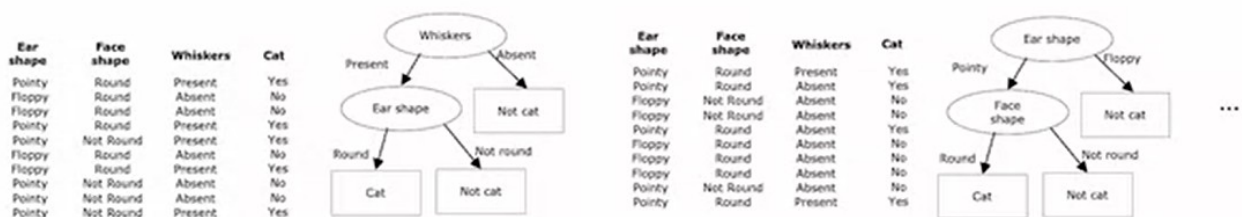
B is the number of decision trees we will plant (generally it is recommended between 64 and 128; from there, the computation cost increases significantly but there is not much improvement for the algorithm):

Given training set of size m

For $b = 1$ to B

Use sampling with replacement to create a new training set of size m

Train a decision tree on the new dataset



For each iteration we create a random set and then we create a decision tree. There is a drawback to this algorithm: it is very common that, even with the sampling with replacement, the feature picked for the root node is almost the same for all trees - or even the same. The same can happen to the nodes near the root node.

To further randomize the trees and get more accurate predictions, we **randomize the feature choice**:

At each node, when choosing a feature to split, if n features are available, pick a random subset of $k < n$ features and allow the algorithm to only choose from that subset of features.

When n is large (tens, or hundreds), a typical choice for the value of k would be: $k = \sqrt{n}$

And now, we have completed the random forest algorithm! It is a much better and robust algorithm than decision trees, as we are averaging lots of small changes, and the result will be less likely to be severely affected by changes in the data.

XGBoost

Today by far the most common implementation of decision tree ensembles is the algorithm XGBoost. It runs quickly, the open source implementations are easily used and has been used very successfully to win machine learning competitions as well as in commercial applications.

The intuition is that, if you want to master playing a particular song, it is much more efficient to focus your effort on the difficult parts that on playing everything over and over again. In a similar way, to improve the algorithm's performance we focus on the misclassified examples from the previous tree, instead of just planting random trees.

Following this idea, this is the **boosted trees implementation**, which is as the random forest implementation but adding a new step on the sampling:

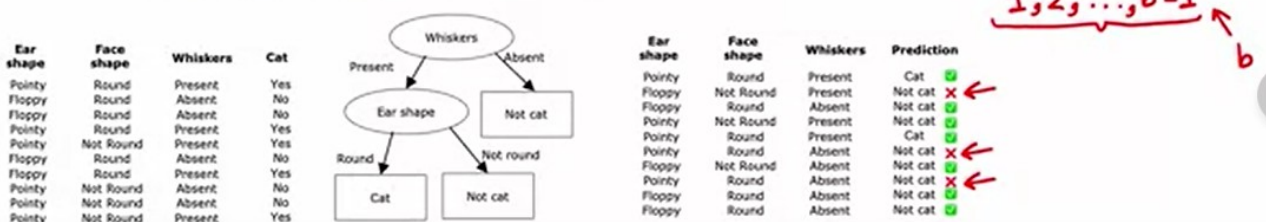
Given training set of size m

For $b = 1$ to B :

Use sampling with replacement to create a new training set of size m

But instead of picking from all examples with equal $(1/m)$ probability, make it more likely to pick misclassified examples from previously trained trees

Train a decision tree on the new dataset



In the upper example we see that, after planting the first tree, there are 3 misclassified examples (the ones with the cross): the algorithm will increase the probability of picking them up for the following samples. Andrew says that how much we increase the probability is a difficult topic, so doesn't go into it.

The most popular implementation of this concept is XGBoost (eXtreme Gradient Boosting):

- Open source implementation.
- Fast and efficient implementation.
- Good choice of default splitting criteria and criteria for when to stop splitting.
- Built in regularization to prevent overfitting.

XGBoost assigns different weights to the examples depending on how successful it was classifying them. The details are quite complex, so practitioners usually use libraries for implementation:

Classification

```
from xgboost import XGBClassifier

model = XGBClassifier()

model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

Regression

```
from xgboost import XGBRegressor

model = XGBRegressor()

model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

When to use decision trees

When going for a decision tree algorithm, Andrew recommends always going for XGBoost.

Decision trees vs neural networks:

Decision Trees and Tree ensembles

- Works well on tabular (structured) data
- Not recommended for unstructured data (images, audio, text)
- Fast
- Small decision trees may be human interpretable

Neural Networks

- Works well on all types of data, including tabular (structured) and unstructured data
- May be slower than a decision tree
- Works with transfer learning
- When building a system of multiple models working together, it might be easier to string together multiple neural networks