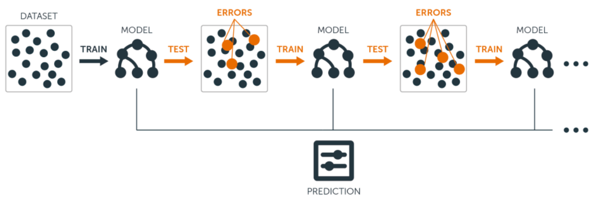
Tree Models
and model ensembles

Machine Learning 2020
mlvu.github.io

gradient boosted decision trees



source: **https://blog.bigml.com/2017/03/14/introduction-to-boosted-trees/**

As a throughline for today’s lecture, we’ll take the method of **gradient boosted decision trees**. This is an algorithm that takes the decision tree model, and uses a method called **boosting**, which combines many different models into one. Gradient boosted tree models are very popular, because they tend to work extremely well out-of-the-box for classical machine learning problems (i.e. problems described by an instance/feature table and a target label).

linear models 2

part 1: Tree Models
Decision trees
Tree models used for classification

Regression trees

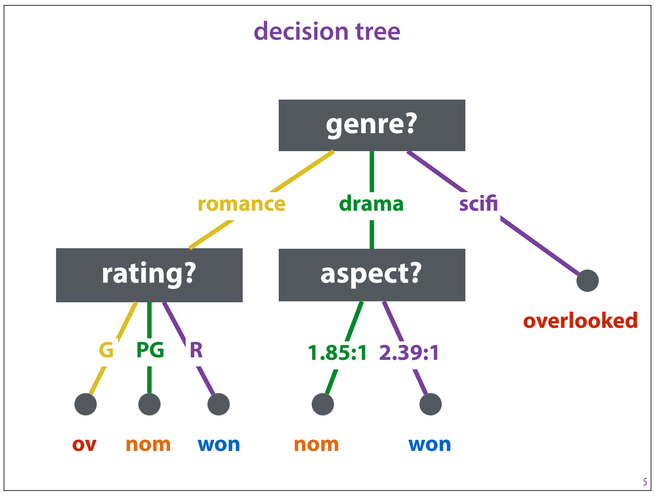
part 2: Ensembling methods
Bagging

Boosting
AdaBoost, Gradient boosting

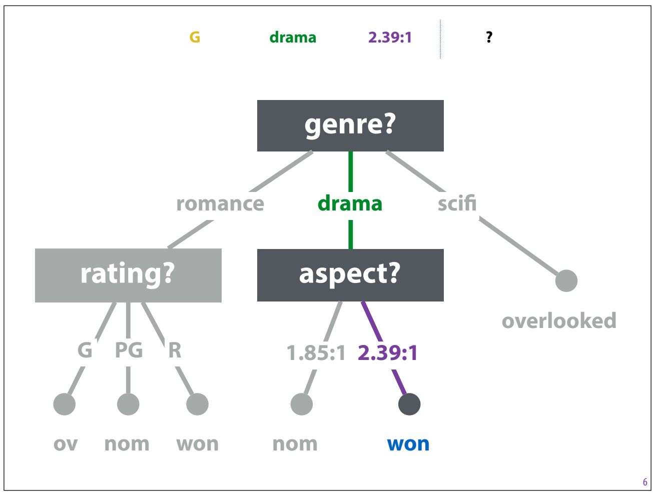
Stacking

rating	genre	aspect ratio	outcome
PG	scifi	1.85:1	overlooked
G	drama	1.85:1	won
G	romance	1.85:1	nominated
R	drama	1.85:1	nominated
G	drama	2.39:1	nominated
G	romance	2.39:1	nominated
R	romance	1.85:1	won
PG	drama	2.39:1	won
PG	scifi	1.85:1	overlooked
G	scifi	2.39:1	overlooked

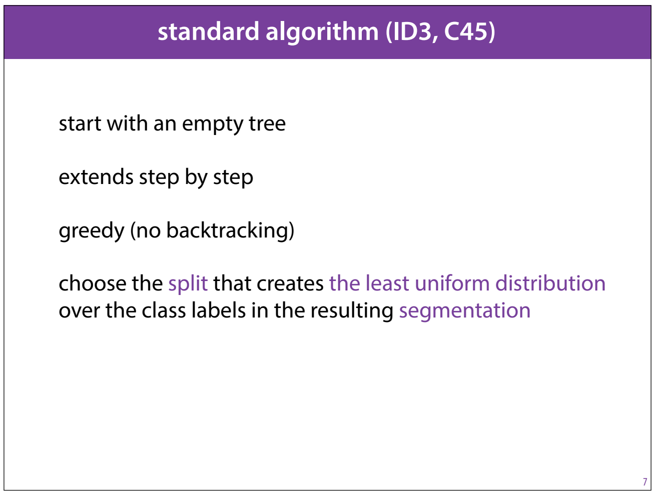
Decision trees in their simplest form work on data with categorical feature.s We’ll use this dataset as a running example: each instance (row) is a movie, and the target class is to predict whether a movie **won** an oscar, was merely **nominated**, or **overlooked**.



This is what a trained decision tree might look like. Each internal node asks the value of a particular feature, and sends the instance to one of its children dependending on the value of that feature.



Here is an example: If we see a movie with a G rating, the genre drama, and a 2.39:1 aspect ratio, we follow the tree to the highlighted leaf node and label the example as **won** (i.e we predict that this movie will win an oscar).



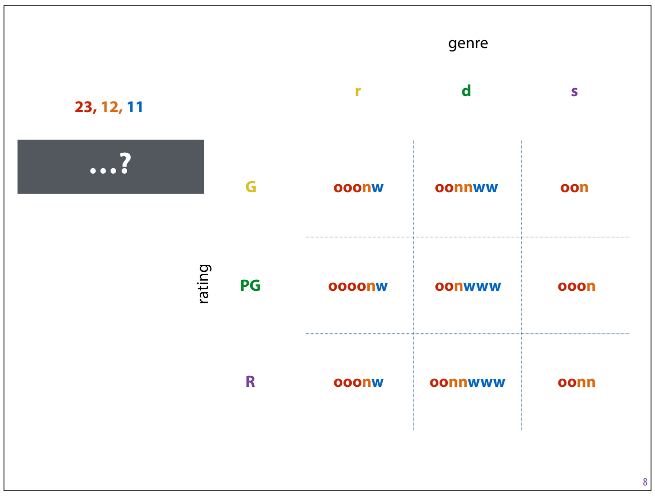
standard algorithm (ID3, C45)

start with an empty tree

extends step by step

greedy (no backtracking)

choose the split that creates the least uniform distribution over the class labels in the resulting segmentation

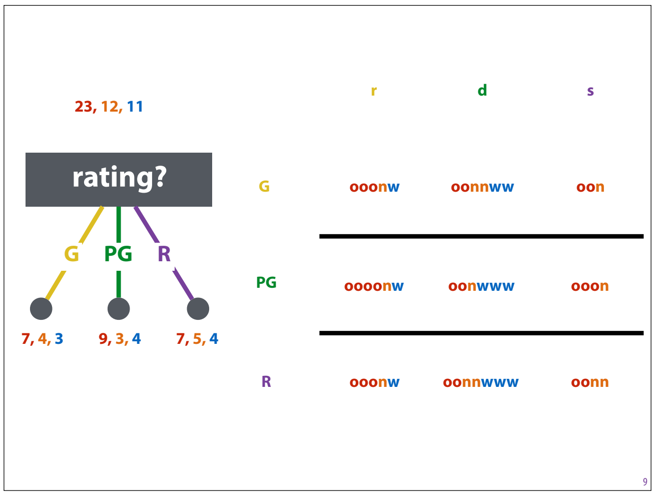


23, 12, 11

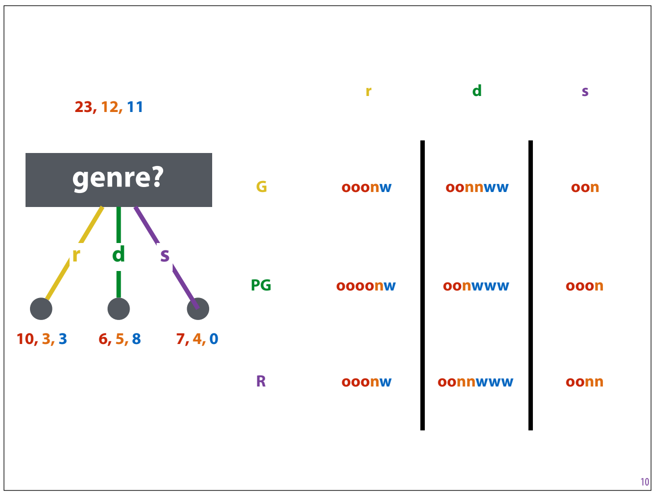
…?

rating \ genre	r	d	s
G	ooonw	oonnww	oon
PG	oooonw	oonwww	ooon
R	ooonw	oonnwww	oonn

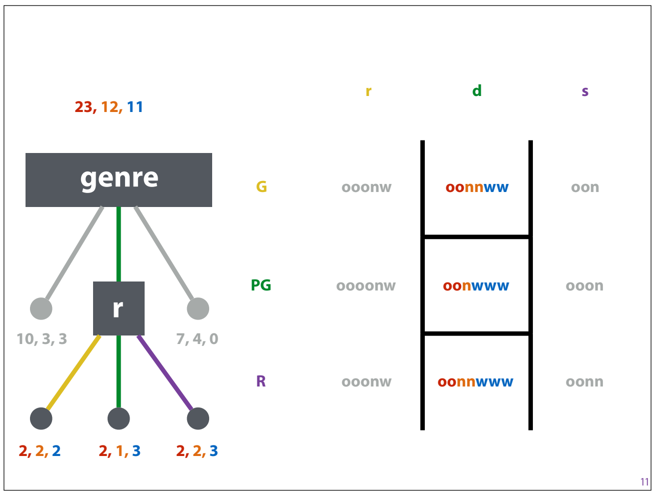
Here we’ve plotted our dated for two features (ignoring the third for the moment). Which of these features provides a better split?



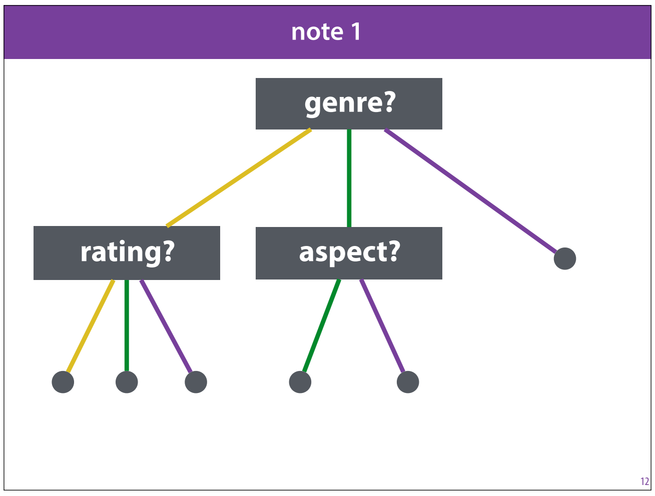
If we split by rating, we get three segments (the three rows on the right). Tallying up the proportions of each class, we see that the proportions of the segments are not that different from the proportions in the whole. In other words, knowing the value of the rating doesn't doesn;doesn't change the information we have about the class very much.



On the other hand, if we split by genre we see that the resulting distributions are much more different: knowing that a movie has the genre **scifi**, allows us to say with near certainty that it won't win an oscar.

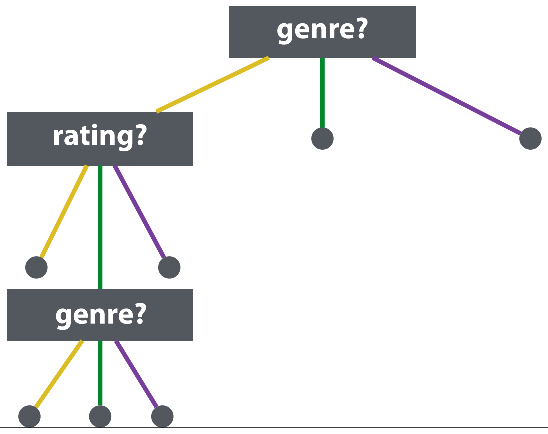


If we split by genre and then by rating, we get this segmentation of the instance space. Each of these regions, corresponding to a leaf node is called a **segment**.



We choose a separate split for each node we extend. For each of the three children of the genre node, we may choose different features to split on.

note 2



13

There's no use (with categorical features) in splitting on the same feature twice. Every instance that encounters the lower genre node will have the yellow genre, so we're not splitting the data at all.

stop conditions

When the maximum depth has been reached.
output label: majority class

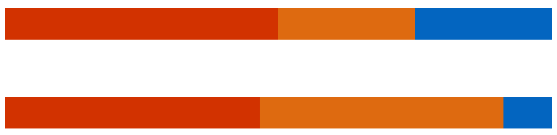
When all feature values are the same
output label: majority class

When all class values are the same
output label: left over class

14

The maximum depth is an optional hyperparameter. We can also train without a maximum depth and rely only on the other two stop conditions.

how do we define "uniformity"



15

In order to make this into a proper algorithm we need to make this more precise. The best feature to split on is the one that creates (averaged over all child nodes) the most non-uniform class distribution in the resulting segment.

How do we measure the non-uniformity of a distribution? It's straightforward for two classes (the further from 50/50 the less uniform), but for more than two classes, it's not so clear cut. Here we see two distributions. In the first, the proportion of the red class is bigger than in the second, but in the second the remainder is divided up between blue and orange in a more non uniform way. Which of these two distributions is less uniform?

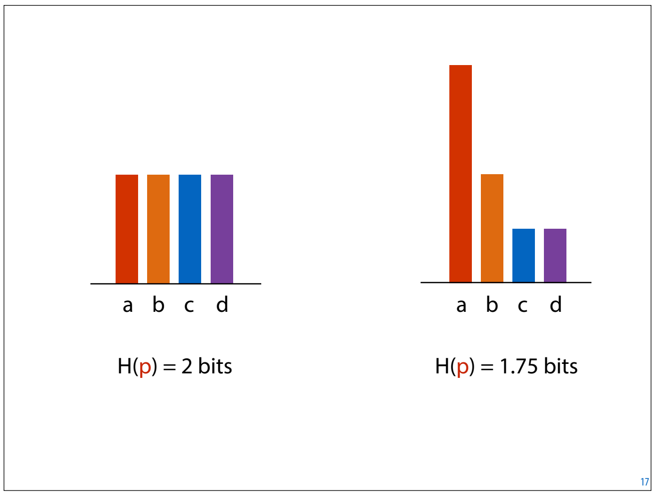
entropy

$p(X)$: data source

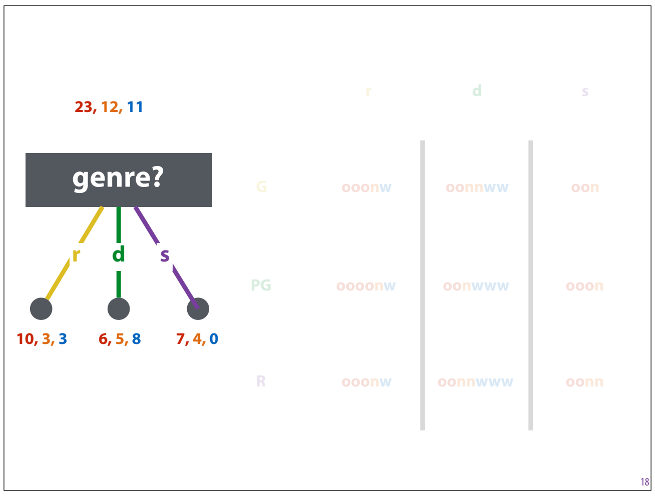
$$\begin{aligned} H(p) &= \sum_{x \in X} p(x) L(x) \\ &= -\sum_{x \in X} p(x) \log p(x) \end{aligned}$$

16

To answer this, we need to look back to the first probability lecture, where we encountered *Entropy*.

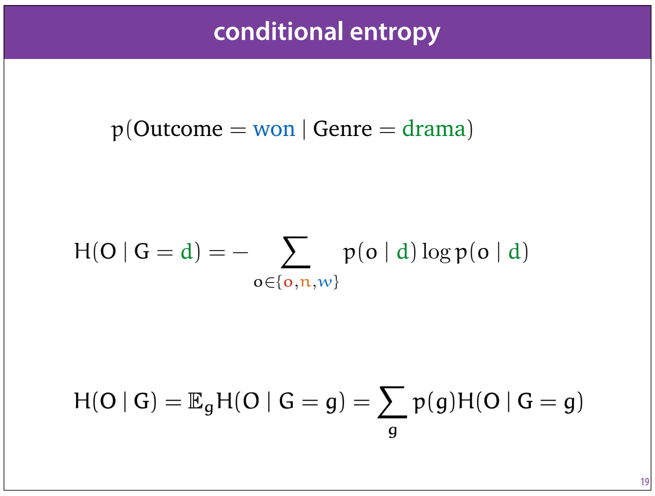


The more uniform our distribution is (the more unsure we are) the higher the entropy.



	r	d	s
G	ooonw	oonnww	oon
PG	oooonw	oonwww	ooon
R	ooonw	oonnwww	oonn

So we can use entropy to establish how uneven a class distribution is, and the more uneven, the better we like the split. But to evaluate this split here, we need to look at four different distributions: the three after the split and the one before (remember, we are evaluating all possible splits over the whole tree, so the incoming distribution may differ between candidates).



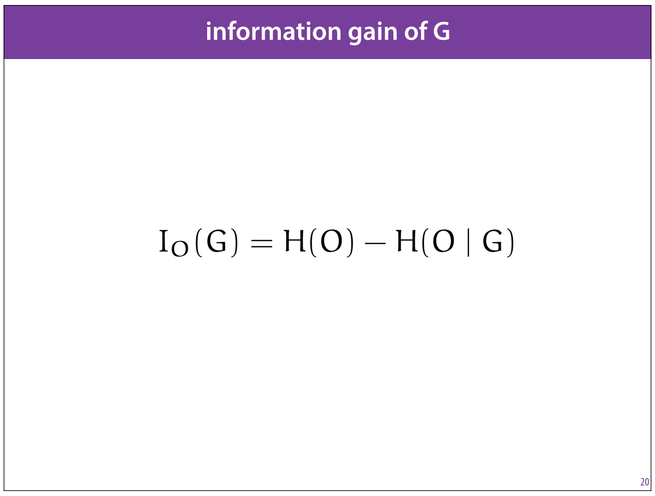
conditional entropy

$$p(\text{Outcome} = \text{won} \mid \text{Genre} = \text{drama})$$

$$H(O \mid G = d) = -\sum_{o \in \{o,n,w\}} p(o \mid d) \log p(o \mid d)$$

$$H(O \mid G) = \mathbb{E}_g H(O \mid G = g) = \sum_g p(g) H(O \mid G = g)$$

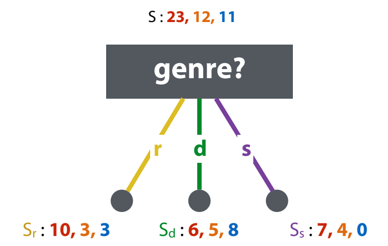
To apply this to the multiple children that a split creates, we can use **conditional entropy**. Conditional entropy is just the entropy of a conditional distribution, summed over all values of the conditional, weighted by the marginal probability of that value.



information gain of G

$$I_O(G) = H(O) - H(O \mid G)$$

The **information gain** measures how much knowing the value of G decreases the entropy of O (i.e. increases what we know about O).



$$I_S(G) = H(S) - \sum_i \frac{|S_i|}{|S|} H(S_i)$$

In practice that gives us this formula, for the information gain of a split. If the set of instances before the split is S, and the split gives us subsets S_i, the information gain is the entropy of S minus the sum of the entropies of the split sets, each weighted by the proportion of instances of S contained in S_i.

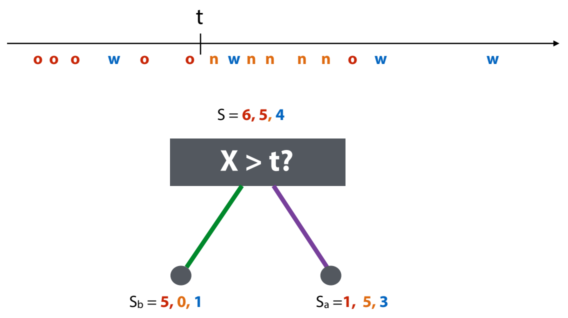
When we compute the entropy of a set like S, we just use the relative frequencies to estimate the probabilities. For instance, we estimate the probability of the **red class** in S as **23**/(**23**+**12**+**11**).

algorithm

```
start with a single unlabeled leaf
loop until no unlabeled leaves:
    for each unlabeled leaf l with segment S:
        if stop condition: label majority class of S
        else: split l on feature F with highest gain $I_S(F)$
```

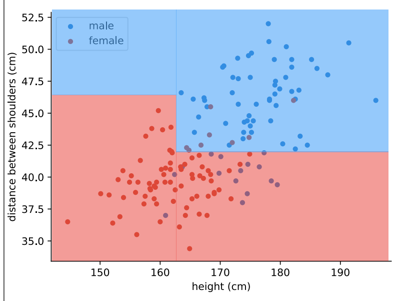
numeric features

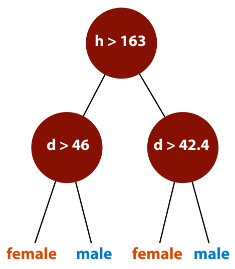
split at *threshold* t



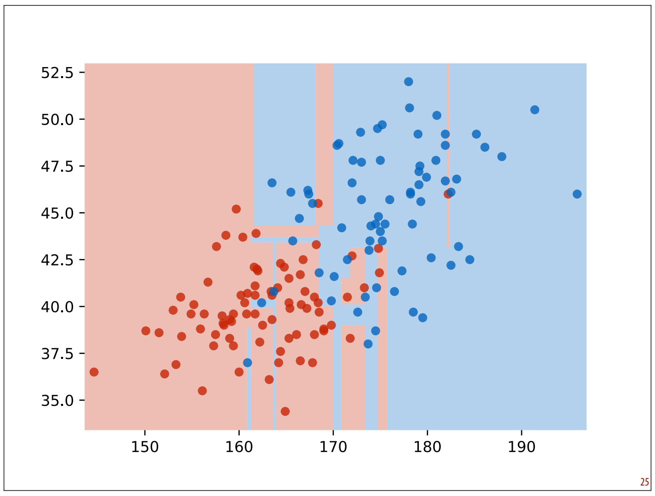
If our dataset contains numeric features, we can deal with this by choosing a threshold t, the node splitting on a numeric features splits the segment in two: the instance for which the feature is lower than t go to one child, and the instances for which the features is higher go to the other.

To compute the optimal threshold we only need to look at numeric values halfway between two instances with a different class. We compute the information gain for each and choose the threshold which provides the highest information gain.





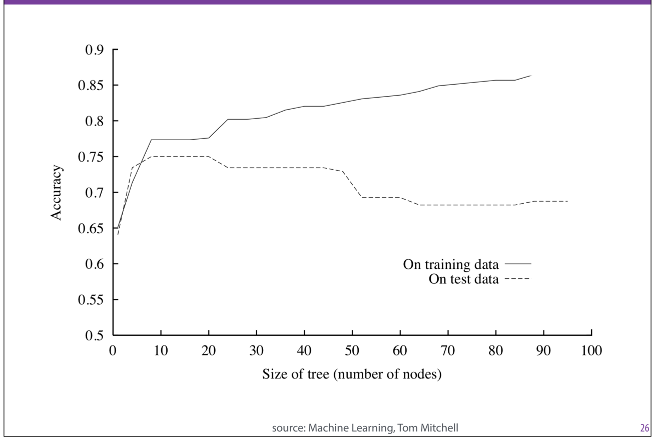
We saw a classifier with numeric features in the opening lecture.



When training an actual decision tree on these two numeric features, we saw quite a complex decision boundary emerge.

This is possible because with numeric features it *does* make sense to split twice on the same feature; we just have to split on a different threshold each time.

overfitting

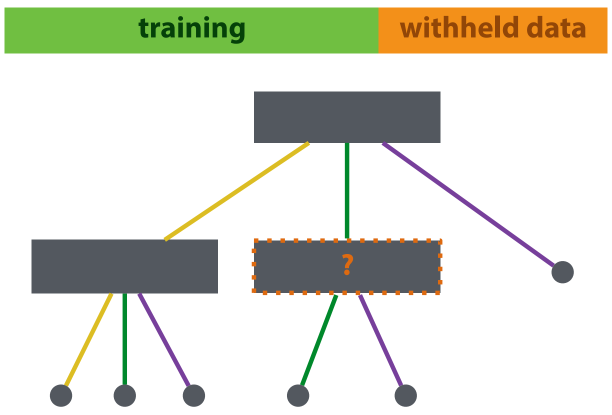


source: Machine Learning, Tom Mitchell

Which brings us to the problem of overfitting. The larger the tree grows, the more likely it is to overfit the data. For this plot the size of the tree is limited to a particular maximum. As the maximum grows, the training accuracy increases, but the test accuracy decreases. A clear sign that the model is overfitting.

Source: Machine Learning, Tom Mitchell

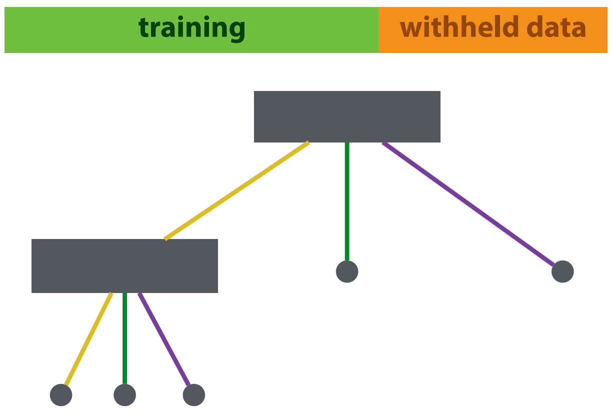
pruning

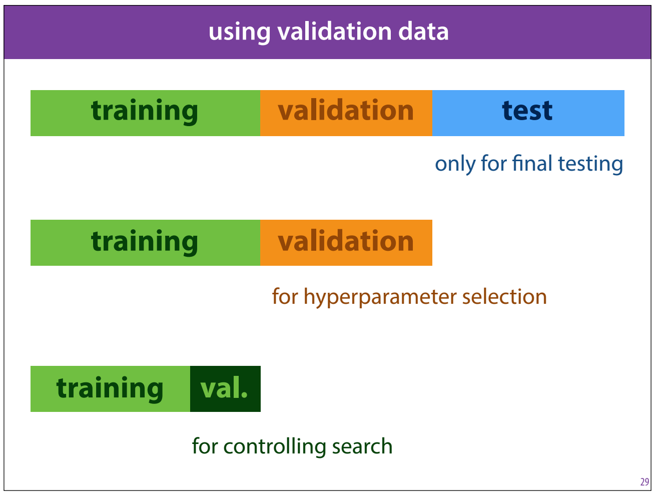


To reduce overfitting, we can **prune** a tree.

After training the full (likely overfitting) tree, we work backwards from the leaves. For each leaf, we check (on withheld data) whether the tree classifier better with the leaf or without. If it works better without, we remove the node.We keep pruning leaves until the performance stops improving.

pruning



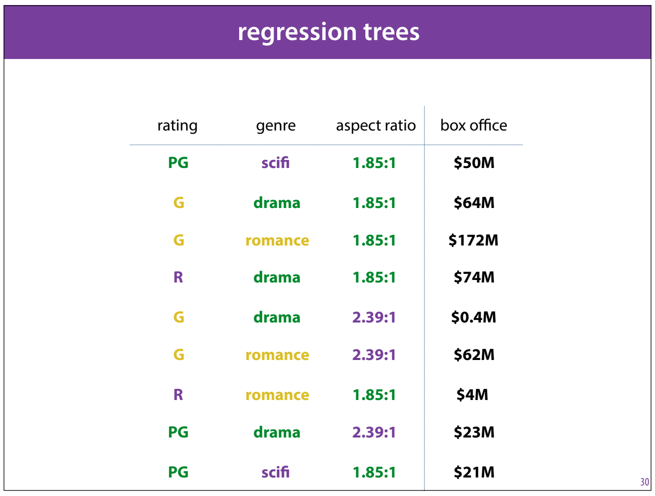


using validation data

It's important to note that when we use a validation set to guide search, that we are using validation data to select our model. This means that if we are also using a validation set, for instance to select whether we'll use a kNN classifier or a decision tree, the pruning should happen on a withheld part of the training data, and not on the same validation data that we use to do our hyperparameter selection.

To see why, imagine what happens when we train our final model (supposing that we've selected a decision tree). During training, we can't use the test set to do our pruning. We can only see the test set when we've decided what our final model is going to be. The first train/validation split should simulate this,situation, so we can't use the orange validation set for controlling search.

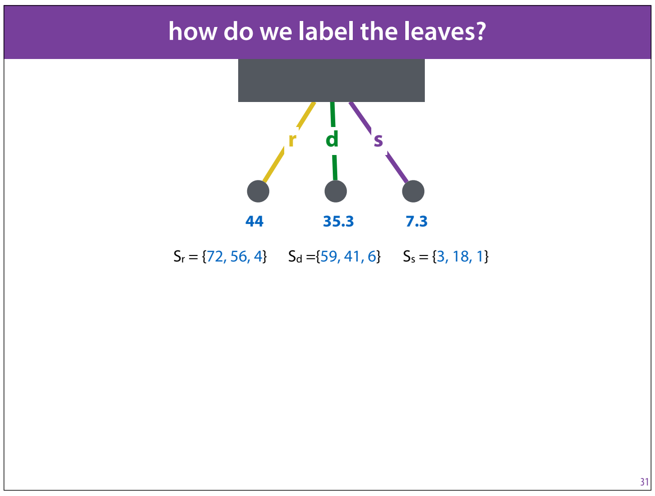
This also goes for *early stopping* in neural networks



regression trees

rating	genre	aspect ratio	box office
PG	scifi	1.85:1	$50M
G	drama	1.85:1	$64M
G	romance	1.85:1	$172M
R	drama	1.85:1	$74M
G	drama	2.39:1	$0.4M
G	romance	2.39:1	$62M
R	romance	1.85:1	$4M
PG	drama	2.39:1	$23M
PG	scifi	1.85:1	$21M

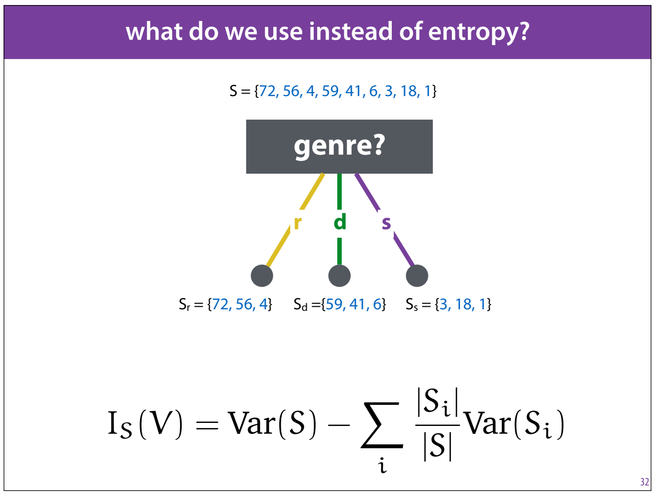
We've seen classification trees that use numerical feature, but what if the target label is numerical? In this case, the model is called a *regression* tree.



how do we label the leaves?

$S_r = \{72, 56, 4\}$ $S_d = \{59, 41, 6\}$ $S_s = \{3, 18, 1\}$

We can label the leaves with the mean or the median of the training instances in the resulting segment.



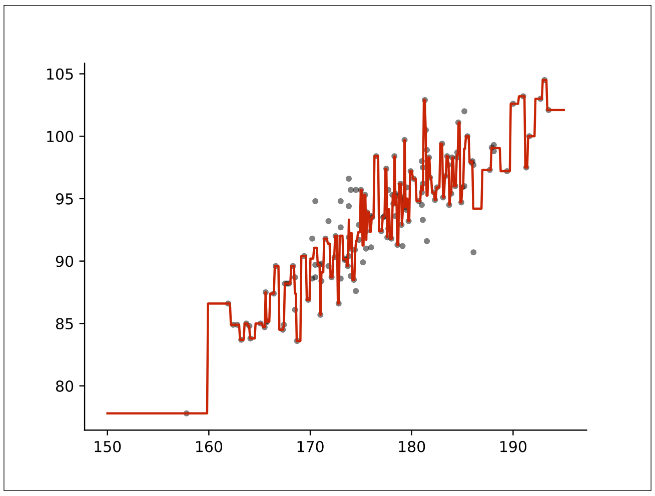
what do we use instead of entropy?

$S = \{72, 56, 4, 59, 41, 6, 3, 18, 1\}$

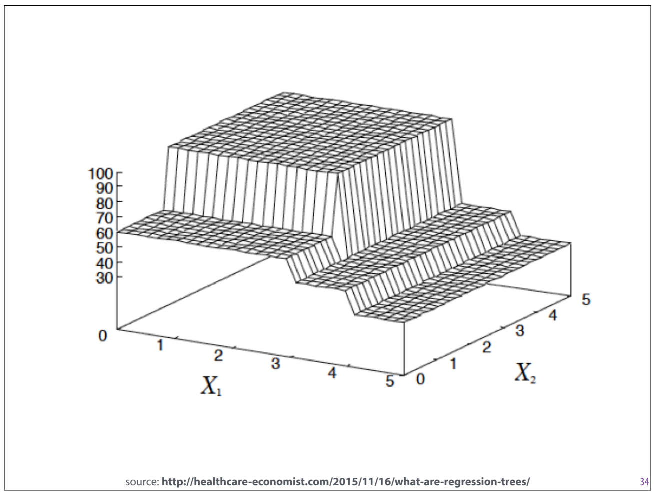
$S_r = \{72, 56, 4\}$ $S_d = \{59, 41, 6\}$ $S_s = \{3, 18, 1\}$

$$I_S(V) = \text{Var}(S) - \sum_i \frac{|S_i|}{|S|}\text{Var}(S_i)$$

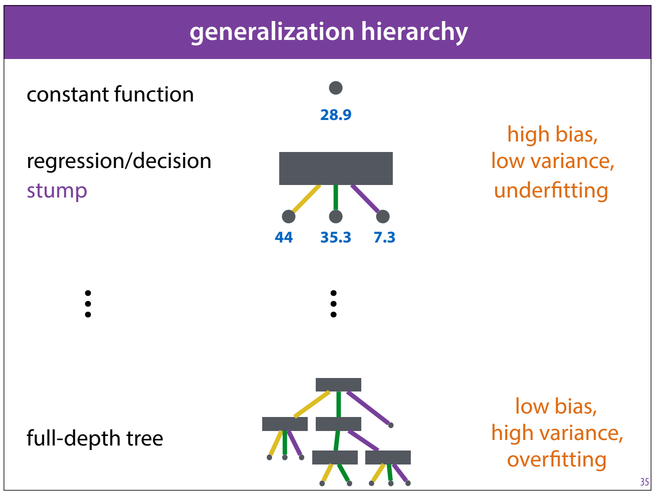
We can't compute entropy over the target values because most likely, they'll all be different. However variance measures a very similar property: the bigger the spread in the set of output labels, the less certain we are about what the value of the leaf node should be. The best split results in a large reduction of average variance over the created segments.



Here's what a regression tree looks like over one numeric feature.



And here's what it looks like for two numeric features.



generalization hierarchy

constant function
28.9
regression/decision stump
44
35.3
7.3
high bias, low variance, underfitting
full-depth tree
low bias, high variance, overfitting

Tree models are a classic example of a model class that provides a **generalization hierarchy**. At one end, the model class provides both very simple, low capacity models like **constant models**, which output just one value for all instances (i.e. a tree without splits) and **stumps**, models that make just one split. These are low capacity models with high bias and low variance, that generalize a lot.

At the other end are full-depth tree models, which are very likely to memorise irrelevant details of the data, and overfit a lot.

decision/regression forests

Single decision trees are not very popular. To make them effective, we need to train many of them and combine them into a single model. These are called decision or regression **forests**, and they're an example of a **model ensemble**, which we'll discuss after the break.

break

bias and variance

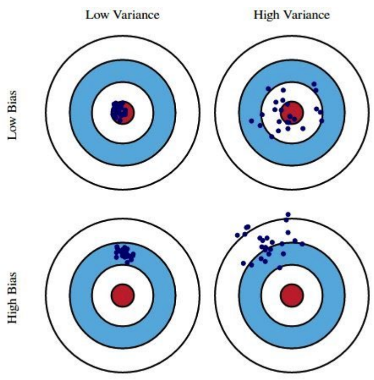
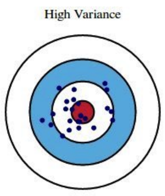


image source: **http://scott.fortmann-roe.com/docs/BiasVariance.html** (recommended reading)

We've talked about bias and variance before. Here is a little visual reminder.



aka. an *unstable* learner

An unstable learner is a learner (a combination of a model with a search algorithm) that has a high variance. It may get a good performance, but slight perturbations in the data can throw that performance off.

In this dartboard metaphor, very dart represent one machine learning experiment on a freshly sampled dataset. In reality, we don't have the luxury of sampling multiple datasets: we only have one dart. If we had multiple datasets, we could train multiple learners and try to combine their judgements into one average judgement.

from methodology 2: bootstrapping

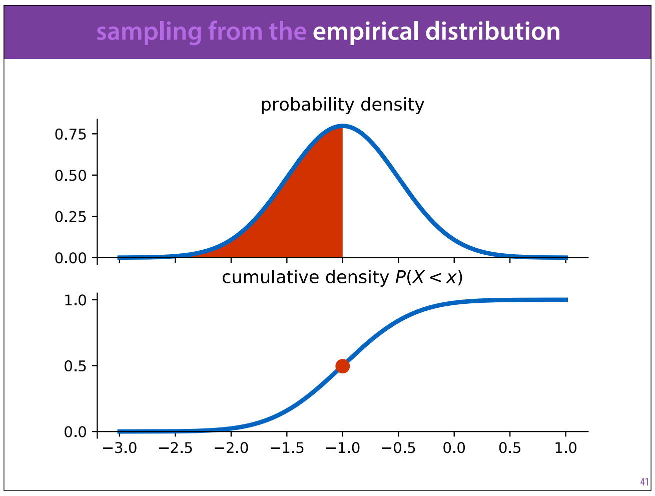
Sample, with replacement, a dataset of the same size as the whole dataset.
On average, about 63.2% of the dataset will be included. The rest will be duplicated instances.

Each bootstrapped sample lets you repeat your experiment.

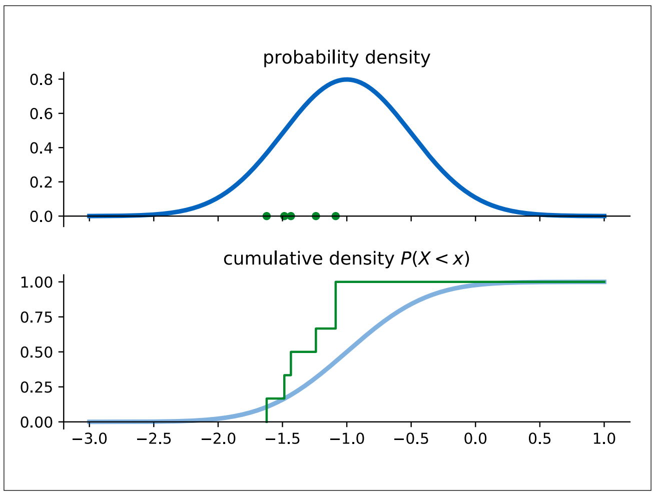
Note that some classifiers will respond poorly to presence of duplicate instances.

Better than cross validation for small datasets.

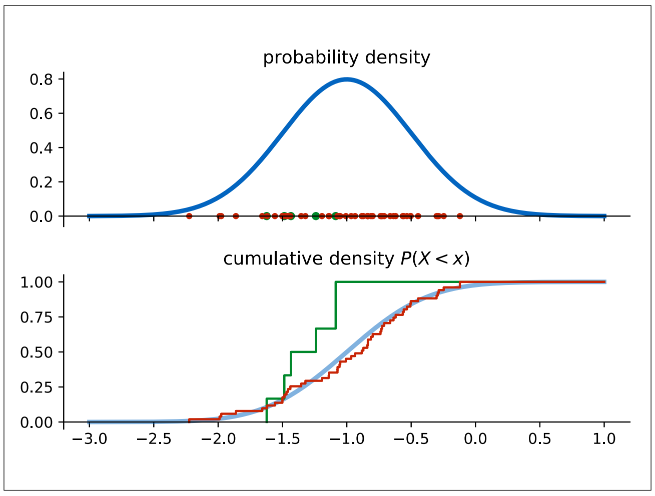
In lecture Methodology 2, we saw a method for simulating the sampling of multiple datasets form the source of our original data: **bootstrapping**.


sampling from the empirical distribution

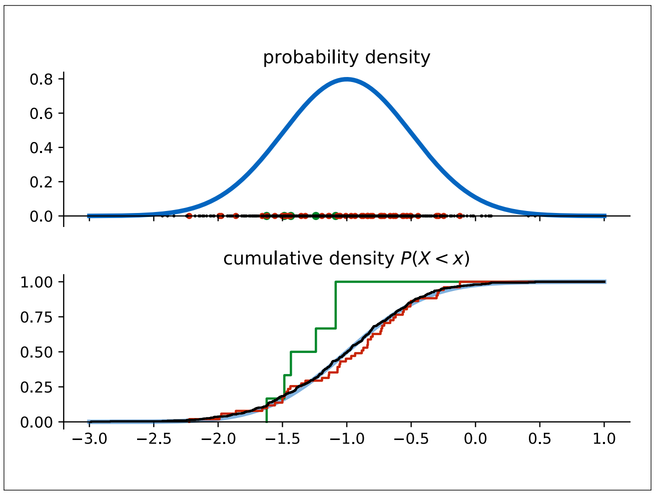
Note that bootstrapping isn't just an ad-hoc method. In a very precise sense, we are sampling from a distribution that approximates the original data distribution. To see how, we'll imagine that we're sampling single scalars from a normal distribution. It's most instructive to look at the **cumulative density function** (CDF).



If we sample 5 points from the original normal distribution, and then sample one point from that dataset, we are essentially sampling from the green CDF. This is called the empirical distribution.



If we increase the size of the original data (to 50 points), we see that the empirical CDF becomes a better approximation of the true CDF

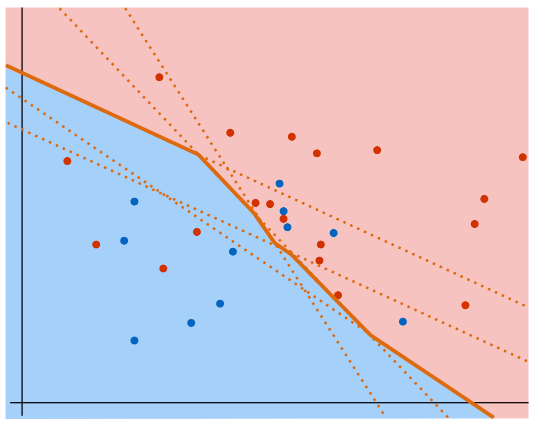


At 500 points, the empirical CDF and the original CDF are almost indistinguishable.

bagging: bootstrap aggreggating

- resample k datasets, and train k models
  This collection is our *ensemble*
- The ensemble classifies by majority vote.
  For class probabilities, use the relative frequency among the votes.

Bagging is our first ensembling method: we resample the data many times through bootstrapping and train a new model for each bootstrapped sample. Then, for a new instance, we classify by taking the majority vote among the ensemble.



source: adapted from *Machine Learning* by Peter Flach, figure 11.1

random forests

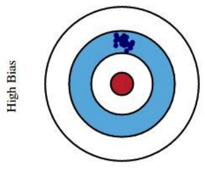
Bagging with decision trees.

Subsample the data *and* the features for each model in the ensemble.

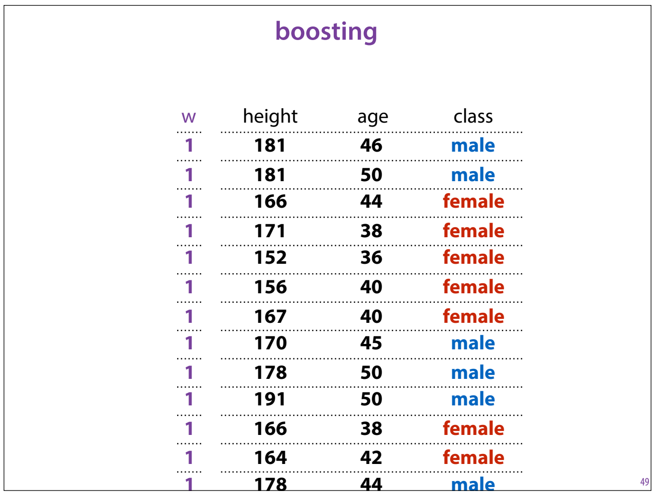
Reduces variance, few hyperparameters, easy to parallelise.

No reduction of bias.

the *hypothesis boosting* question



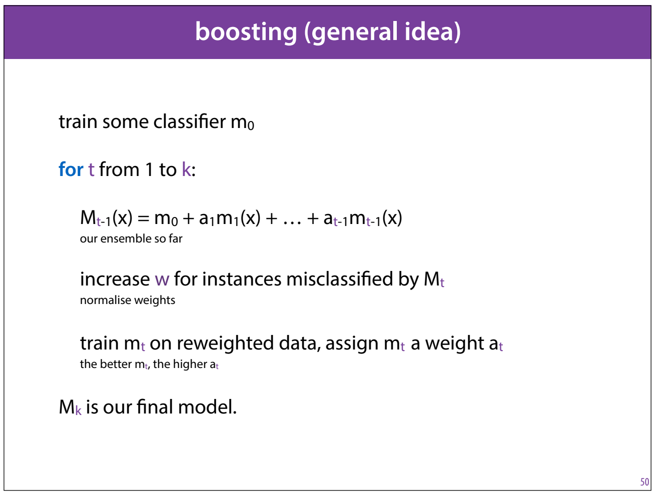
Bagging helps for models with high variance and low bias. If we have the opposite, a model family with high bias and low variance, can we achieve the same? This is the *hypothesis boosting* question (hypothesis is a synonym for model family, here). Is there an ensembling method that allows us to create a series of models from a family with high bias and to create an ensemble that together has low bias (possibly at the expense of a higher variance).



boosting

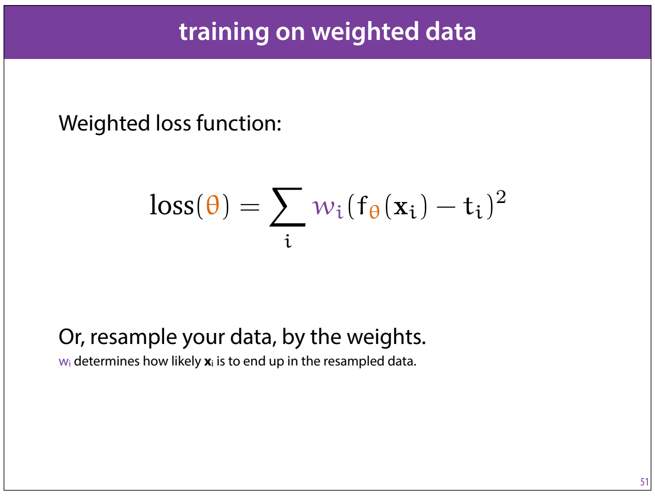
w	height	age	class
1	181	46	male
1	181	50	male
1	166	44	female
1	171	38	female
1	152	36	female
1	156	40	female
1	167	40	female
1	170	45	male
1	178	50	male
1	191	50	male
1	166	38	female
1	164	42	female
1	178	44	male

Most boosting methods work by adding a weight to each instance in the data. For each new model, we lower the weights of the points that the previous models got right, and increase the weight of the points that the previous models got wrong. We then train the next model to focus on this reweighed version of the data.



boosting (general idea)

```
train some classifier $m_0$

for t from 1 to k:

    $M_{t-1}(x) = m_0 + a_1 m_1(x) + \ldots + a_{t-1} m_{t-1}(x)$
    our ensemble so far

    increase w for instances misclassified by $M_t$
    normalise weights

    train $m_t$ on reweighted data, assign $m_t$ a weight $a_t$
    the better $m_t$, the higher $a_t$

$M_k$ is our final model.
```



training on weighted data

Weighted loss function:

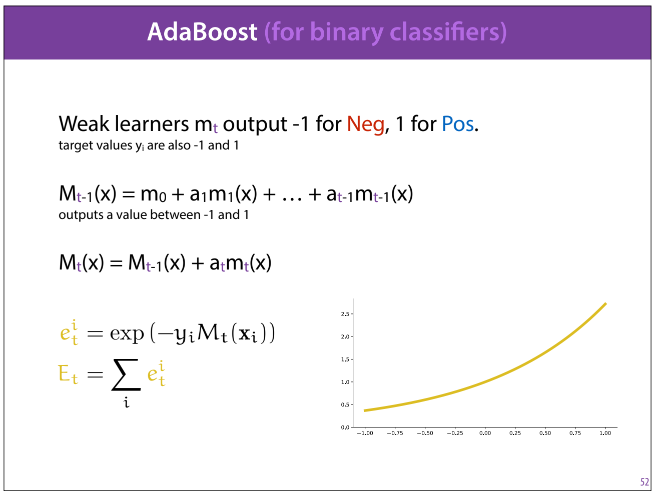
$$\text{loss}(\theta) = \sum_i w_i (f_\theta(\mathbf{x}_i) - t_i)^2$$

Or, resample your data, by the weights.

w_i determines how likely $\mathbf{x}_i$ is to end up in the resampled data.

How do we train on a weighted dataset? If we have a loss function, we can just make the sum over the loss of each instance a weighted sum.

If our model doesn't have an explicit loss function (like regression trees), we can resample the dataset, and make the instance with higher weights more likely to be sampled.



AdaBoost (for binary classifiers)

Weak learners m_t output -1 for Neg, 1 for Pos.

target values y_i are also -1 and 1

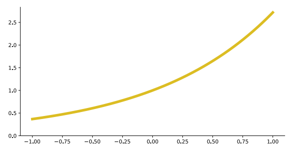
$M_{t-1}(x) = m_0 + a_1 m_1(x) + \ldots + a_{t-1} m_{t-1}(x)$

outputs a value between -1 and 1

$M_t(x) = M_{t-1}(x) + a_t m_t(x)$

$$e^i_t = \exp\left(-y_i M_t(\mathbf{x}_i)\right)$$

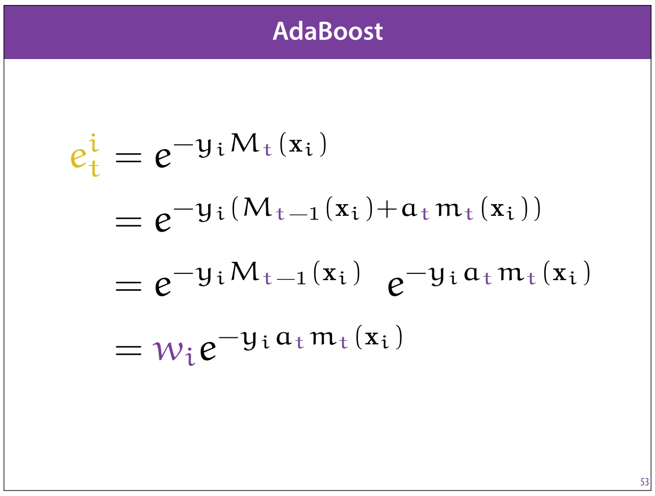
$$E_t = \sum_i e^i_t$$



We'll first take a look at **AdaBoost** (which stands for adaptive boosting), which is a principled derivation for how to set the weights w_i, and and the model weight a_t.

We assume we've already trained the ensemble up to model m_{t-1}, giving us ensemble model M_{t-1}. We now need to make two choices: which model m_t to choose (or what loss to minimize when training this model) and which model weight a_t to assign it.

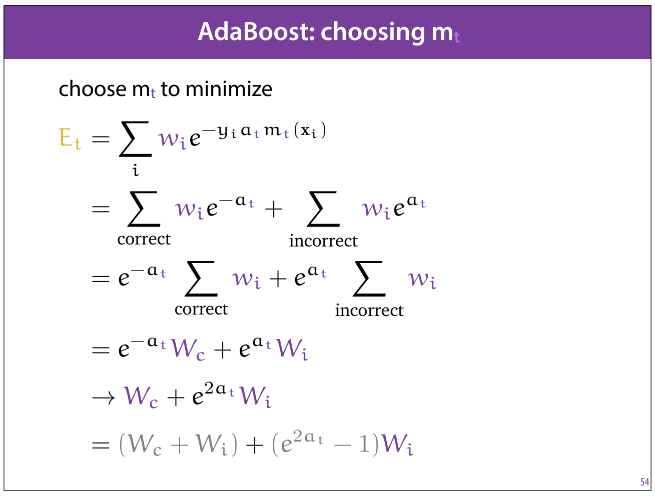
To do so, we first define the error (aka the loss) for the ensemble at step t, and then choose m_t and a_t to minimize this loss.



AdaBoost

$$
\begin{aligned}
e_t^i &= e^{-y_i M_t(x_i)} \\
&= e^{-y_i (M_{t-1}(x_i) + a_t m_t(x_i))} \\
&= e^{-y_i M_{t-1}(x_i)} \; e^{-y_i a_t m_t(x_i)} \\
&= w_i e^{-y_i a_t m_t(x_i)}
\end{aligned}
$$

We can rewrite the per-instance loss to separate the error cause by the ensemble so far, and the loss caused by our choice of model m_t. The first is a constant (since the ensemble up to m_{t-1} has already been chosen), which becomes the weight w_i of instance $\mathbf{x}_i$.



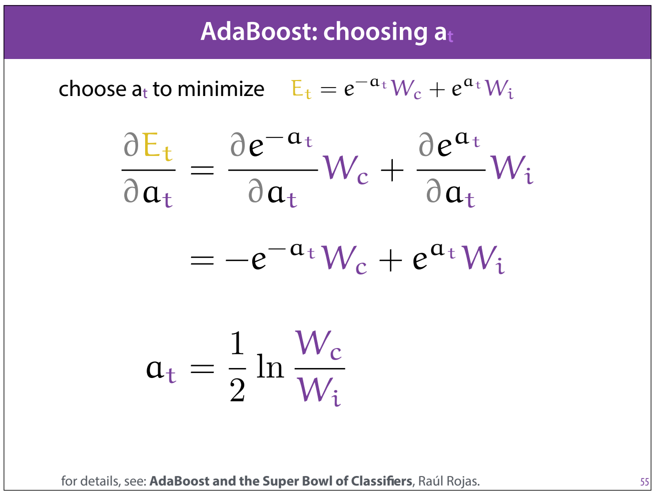
AdaBoost: choosing m_t

choose m_t to minimize

$$
\begin{aligned}
E_t &= \sum_i w_i e^{-y_i a_t m_t(x_i)} \\
&= \sum_{\text{correct}} w_i e^{-a_t} + \sum_{\text{incorrect}} w_i e^{a_t} \\
&= e^{-a_t} \sum_{\text{correct}} w_i + e^{a_t} \sum_{\text{incorrect}} w_i \\
&= e^{-a_t} W_c + e^{a_t} W_i \\
&\to W_c + e^{2a_t} W_i \\
&= (W_c + W_i) + (e^{2a_t} - 1) W_i
\end{aligned}
$$

For the total loss, we can split the sum into the instance that are correctly classified and the instances that are incorrectly classified. This means the exponents become constants in the sum and can be move to the front. Line five is not an equal function, but minimising line four is the same as minimising line 5. In the last line, all the greyed out parts are constant with respect to m_t: W_c and W_i each depend on which classifier we choose, but their sum is just the sum of all the weights provided by the ensemble so far.

The take-away here is that choosing m_t to minimize the error E_t, consists of just minimising the sum of the weights of the instances misclassified by m_t.



AdaBoost: choosing a_t

choose a_t to minimize $E_t = e^{-a_t} W_c + e^{a_t} W_i$

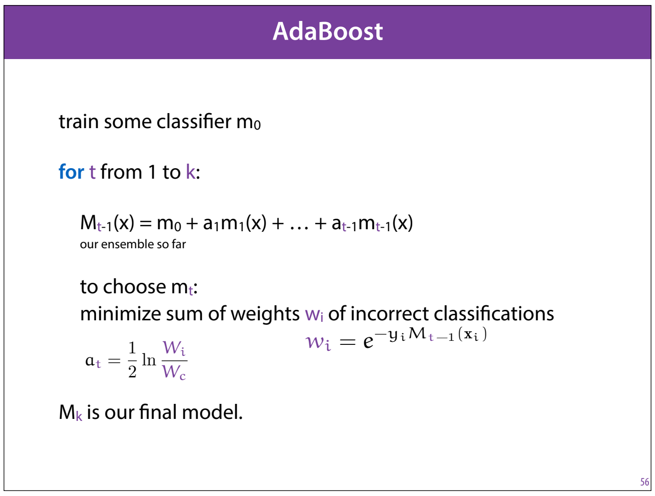
$$
\begin{aligned}
\frac{\partial E_t}{\partial a_t} &= \frac{\partial e^{-a_t}}{\partial a_t} W_c + \frac{\partial e^{a_t}}{\partial a_t} W_i \\
&= -e^{-a_t} W_c + e^{a_t} W_i
\end{aligned}
$$

$$
a_t = \frac{1}{2} \ln \frac{W_c}{W_i}
$$

for details, see: **AdaBoost and the Super Bowl of Classifiers**, Raúl Rojas.

To choose at, we just compute the derivative of the error wrt to a_t, set it to zero and solve for a_t.

Intuitively, the formula for a_t states that the better the proportion of correct to incorrect labelings, the more model t should weigh in the ensemble. The logarithm ensures a kind of diminishing return of weight: getting 11 instances correct instead of 10 had much more impact on the weight than getting 101 instances correct instead of 100.



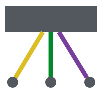
AdaBoost

```
train some classifier $m_0$

for t from 1 to k:

    $M_{t-1}(x) = m_0 + a_1 m_1(x) + \ldots + a_{t-1} m_{t-1}(x)$
    our ensemble so far

    to choose $m_t$:
    minimize sum of weights $w_i$ of incorrect classifications
                                        $w_i = e^{-y_i M_{t-1}(x_i)}$
    $a_t = \frac{1}{2} \ln \frac{W_i}{W_c}$

$M_k$ is our final model.
```

boosting

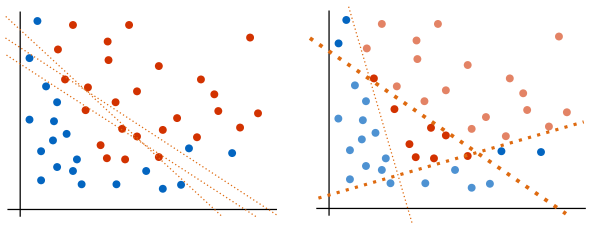
works even if M only classifies just slightly better than chance

- i.e. decision *stumps*



variants: AdaBoost, LogitBoost, BrownBoost

bagging vs boosting



If we visualise the learners, we can see clearly why boosting is so much more powerful than bagging. Since bagging works in parallel, every member of the ensemble will end up looking roughly the same, providing little variation in the ensemble. In boosting, since each learner is trained in sequence, based on what the previous learners did, we get much more variation.

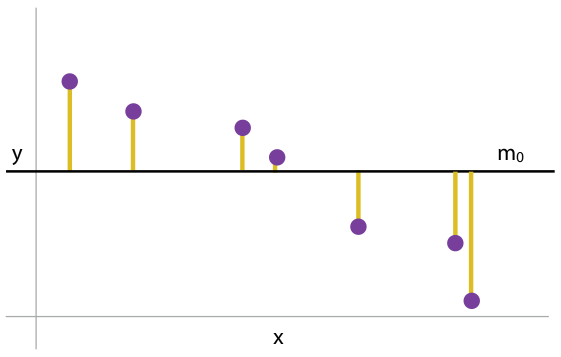
Adapted from Machine Learning by Peter Flach, figure 11.2

gradient boosting

Boosting for *regression* models

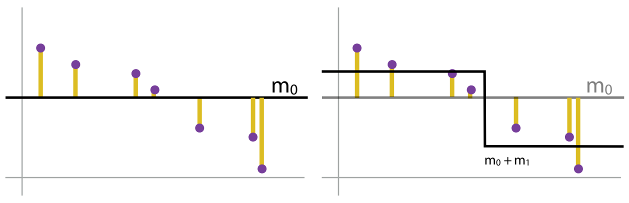
Main intuition: fit the next model to the residuals of the current ensemble.

gradient boosting



To illustrate let’s start with a constant model.

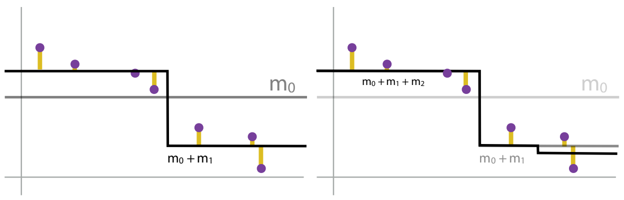
gradient boosting (informally)



m_1 predicts the residuals of m_0

We take the residuals of the previous model, and train a new model, m1, to predict the residuals. The new model adds these predictions to the predictions of the first.

gradient boosting (informally)



This combined model has new residuals, which we can then train another model m2 to predict, add that to the model. and so on.

gradient boosting (for least squares loss)

```
initial model $M_0(\mathbf{x}) = c$

for t from 1 to k:

    for all i: $r_i \leftarrow M_{t-1}(\mathbf{x}_i) - y_i$
    $r_i$ are the residuals of the ensemble so far

    fit model $m_t$ to dataset $\{\mathbf{x}_i, r_i\}$

    $M_t(\mathbf{x}) = M_{t-1}(\mathbf{x}) + \gamma_t m_t(\mathbf{x})$
    $\gamma_t$: optimise by through *line search*, or just slowly decay
```

$$M_3(x) = M_2(x) + \gamma_3 m_3(x)$$

$$= M_1(x) + \gamma_2 m_2(x) + \gamma_3 m_3(x)$$

$$= M_0(x) + \gamma_1 m_2(x) + \gamma_2 m_2(x) + \gamma_3 m_3(x)$$

why *gradient* boosting?

Imagine a model which simply stores a single output for each instance

$$f_w(\mathbf{x}_i) = w_i$$

$$\frac{\partial \frac{1}{2}(f(\mathbf{x}_i) - y_i)^2}{\partial w_i} = w_i - y_i$$

To see why we call it gradient boosting, imagine a model which simply stores a single value w_i which is the value it will predict for instance $\mathbf{x}_i$. The gradient of this model with respect to weight w_i is just the residual of instance i.

Normal models have a more constrained parameter space, but essentially, gradient boosting is telling the model to approximate this gradient in prediction space, as well as possible.

why *gradient* boosting?

$$r_i = \frac{\partial \frac{1}{2}(M(x_i) - y_i)^2}{\partial M(x_i)} = M(x_i) - y_i$$

In other words, the residuals are the gradient of the model output for instance i with respect to the loss for instance i.

pseudo-residuals

$$r_i = \frac{\partial \text{loss}(M(x_i), y_i)}{\partial M(x_i)}$$

This allows us to generalise the idea of gradient descent to other loss functions. We can simply work out this derivative, and train the next model in our ensemble to predict it. The resulting value isn't as intuitive as a proper residual, hence the name **pseudo-residual**, but training the next model to predict the pseudo residuals works just as well to minimize the loss.

for instance: MAE loss

$$\text{loss}(m_i, y_i) = |m_i - y_i|$$

$$\begin{aligned} r_i &= \frac{\partial |m_i - y_i|}{\partial m_i} \\ &= \frac{\partial |m_i - y_i|}{\partial m_i - y_i} \frac{\partial m_i - y_i}{\partial m_i} \\ &= \text{sign}(m_i - y_i) \end{aligned}$$

For instance, here is how it works for the **mean absolute error** loss (also known as the L1 loss): the absolute value of the different between output and target (as opposed to the square for the MSE loss).

Working this out, we see that if we want to use gradient boosting to minimize the MAE loss, we should train the next model in our ensemble to predict the sign (-1 or +1) of the residuals of the ensemble so far.

gradient boosting vs AdaBoost

Gradient boosting

Each model fits the (pseudo) residuals of the previous model.

The ensemble optimises a global loss.
Even if the individual models don't optimise a well-defined loss

AdaBoost

Each model fits a reweighted dataset.

Each model refines its own reweighted loss.

stacking

height	age		M1:	M2:	M3:
181	46	m	m	m	f
181	50	m	m	f	f
166	44	f	f	f	f
171	38	f	m	f	f
152	36	f	f	m	m
156	40	f	m	f	f
167	40	f	f	f	f
170	45	m	m	f	m
178	50	m	m	m	m
191	50	m	m	f	f
166	38	f	f	m	m
164	42	f	f	m	f
178	44	m	m	m	m

Finally: stacking. Stacking is a method that we can use when we already have a small number of model that we'd like to combine into a single model. We simply compute the outputs of these models for every point in the dataset…

stacking

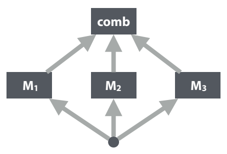
height	age	M_1	M_2	M_3	
181	46	m	m	f	m
181	50	m	f	f	m
166	44	f	f	f	f
171	38	m	f	f	f
152	36	f	m	m	f
156	40	m	f	f	f
167	40	f	f	f	f
170	45	m	f	m	m
178	50	m	m	m	m
191	50	m	f	f	m
166	38	f	m	m	f
164	42	f	m	f	f
178	44	m	m	m	m

..and add them to the dataset as a column.

We then train a new model, called a combiner, on this new, extended data. The combiner can choose to how to combine the "expert advice" of the original models, and it can even use the original features to learn which expert to listen to in which part of the feature space.

combiner: usually logistic regression

If NNs are used for the ensemble, the whole thing becomes one big neural network.
We can even refine the whole ensemble end-to-end using backpropagation



summary: ensembling

Used in production (and competitions) to achieve extra performance on top of a particular model.

Never used in research. We know we can improve any model by boosting.

Can be expensive for big models. Boosting with large deep learning models is rare in production.
Still happens in Kaggle competitions.

Bagging reduces *variance*, boosting reduces *bias*.

mlcourse@peterbloem.nl